CONTRACTOR REPORT ARPAD-CR-80001

DISTRIBUTED SENSOR SYSTEMS AND ELECTROMECHANICAL ANALOG FACILITY

R. A. VOLZ
S. L. BEMENT
R. JUNGCLAS
T. ROSENBAUM
E. J. SESEK
J. WENSTRAND
S. CAGLIASTRO
A. ZEMON
VERCHRON SYSTEMS, INC.
SALINE, MICHIGAN

P. BECK
ARRADCOM, PROJECT ENGINEER

JANUARY 1980

US ARMY ARMAMENT RESEARCH AND DEVELOPMENT COMMAND
PRODUCT ASSURANCE DIRECTORATE
DOVER, NEW JERSEY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Date Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER Contractor Report ARPAD-CR-80001	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALDG NUMBER Final	
4. TITLE *(and Subtitle)* DISTRIBUTED SENSOR SYSTEMS AND ELECTRO-MECHANICAL ANALOG FACILITY	5. TYPE OF REPORT & PERIDD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHDR(s) R.A.Volz, S.L. BeMent, R.Jungclas, T.Rosenbaum, E.J.Sesek, J.Wenstrand, S.Cagliastro, A.Zemon, VerChron Systems, Inc., P.Beck, ARRADCOM	8. CDNTRACT DR GRANT NUMBER(s) DAAG29-76-D-0100
9. PERFORMING ORGANIZATION NAME AND ADDRESS VerChron Systems 325 Tamarack Drive Saline, MI 48176	10. PRDGRAM ELEMENT, PRDJECT, TASK AREA & WDRK UNIT NUMBERS Materials Testing Technology Program
11. CONTROLLING OFFICE NAME AND ADDRESS ARRADCOM, TSD STINFO (DRDAR-TSS) Dover, NJ 07801	12. REPORT DATE January 1980
	13. NUMBER OF PAGES 502
14. MONITDRING AGENCY NAME & ADDRESS*(if different from Controlling Office)* ARRADCOM, PAD SASD (DRDAR-QAA) Dover, NJ 07801	15. SECURITY CLASS. *(of this report)* Unclassified
	15e. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Distributed sensor systems	Distributed computing system
Real time computer applications	Interprocessor communications
Software validation	Concurrent operations
Software test bed	Hierarchical computer systems

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Distributed sensor systems are key ingredients in many real world applications. Specific instances abound both in industrial and military environments, e.g., the monitoring (and possibly control) of manufacturing operations, or the dispersion of various types of sensors to detect enemy movements. There are two major areas of study in distributed sensors: the design and development of the sensors themselves, and the logical use of such sensors. This report is

(continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Date Entered)*

20. ABSTRACT (Continued)

directed toward mechanisms to study the latter. Some examples of the latter, explained in detail, are based on CICE/ECE/IOE 469, a course in Real Time Computing Systems developed at the University of Michigan, Ann Arbor. Both student and faculty critiques of the electromechanical analog facility used in the program are included.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Distributed Sensor Problems

Distributed sensor systems are key ingredients in many real world applications. Specific instances abound both in industrial and military environments, e.g., the monitoring (and possibly control) of manufacturing operations, or the dispersion of various types of sensors to detect enemy movements. There are two major areas of study in distributed sensors: the design and development of the sensors themselves, and the logical use of such sensors. This report is directed toward mechanisms to study the latter.

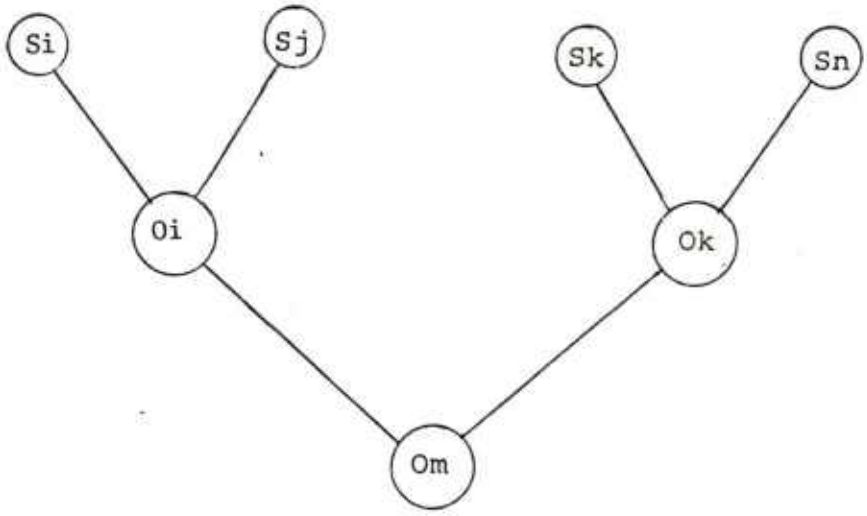
Although a wide variety of sensors and applications exist, there are a number of basic problems common to most systems. First, a sensor only detects an event and/or indicates a piece of information. Accordingly, our model of a sensor will be a generator of a piece of data. There must be one or more observers (human or machine) to record and use that information, and in many instances there may also be some aspect of control in the use of the sensor (e.g., when to take a sample, or reset instrumentation after an event detection). Our sensor model then is a data generating device which can operate either asynchronously upon some event occurrence or upon the command of some observer. Our sensor model may include a control input from the observer.

In distributed sensor systems the overall system architecture, or structure, is an important issue. This problem includes not only the assignment of a sensor to an observer, but the interconnection between observers and direction of data flow as well. Indeed, some observers may operate on data received from the sensor before passing it on to other observers. Figure 1 illustrates a hierarchical interconnection system; systems of this type are very common and arise naturally in many organizations in which the observers are human. There are, however, many other forms of sensor system structures.

Another important issue is the concurrency of data arrival. It may occur either at the first order observer level (the observers actually observing the sensors) or at any higher level of observer interconnection. Various hardware and software arbitration schemes can be developed to deal with these.

There is also the issue of the action to be taken by each observer upon receiving a piece of data from a sensor or lower level observer. In a simple monitoring system the observer may merely aggregate and record the data received, while in others, e.g., an automated assembly line, there may be a complex sequence of actions undertaken, based upon the data, to effect the individual operations being monitored and controlled. The specific actions to be undertaken are highly application

dependent, and not necessarily part of the sensor system (though they are clearly closely related). For purposes of this analysis, specific actions will not be of concern. Rather, the opportunity to perform actions in a timely fashion will be considered.



Hierarchical model of distributed sensor system. Arrows denote permissible direction of data flow. Si are sensors and Oi are observers.

Figure 1.

Finally, there is the issue of timing considerations. In many distributed sensor systems there are either time critical observations to be made (to avoid useless data) or time critical actions (e.g., remove a part from a conveyer before it falls off the end or rams another device). The concept of "real time" operation is then critical to a large number of distributed sensor systems.

1.2 Sensor System Analog

In the study of any complex real system it is often difficult, too costly, or impossible to have the actual system available for study. Rather, the analysis and design generally proceed through several phases of development using mathematical and simulation techniques. Often, preliminary work is done using

mathematical techniques. However, in large complex systems, the mathematics of analysis and design often become computationally intractable. In such cases the use of simulation is quite common. Simulation is also often useful for testing ideas or trial designs without the expense or risk of using the actual system.

One form of simulation is the construction of a device with characteristics similar to the system being studied, but being much less costly, dangerous or difficult to use. Simulation of this type are also often used for training purposes or to assist humans in visualizing the entirety of a complex system. One of the most prominent examples of this form of simulation is the aircraft simulator used to train airline pilots before they try actual flight.

The simulator, or analog, must have certain characteristics in order to be useful for simulation. Most importantly, it must be capable of representing the actual system (i.e., there must be a mapping between the analog and the actual system). Secondly, it must have some quality which makes it more desirable to work with than the actual system, e.g., less costly, less dangerous, smaller, or easier visualization of system behavior. In our view of distributed sensors we desire the analog to possess the following properties:

1. sensors capable of being activated by external events and notifying the corresponding observer that an event has occurred.

2. a mechanism for observer actions, based upon the event occurrence, in "real time".

3. a mechanism for emulating various queuing strategies for concurrent events.

4. capability of simulating hierarchical levels of observer activity.

1.2.1 Computer Controlled Train

A computer controlled N-gauge model railroad is the specific analog used in this study. The track layout is divided into a number of track sections, each separately under computer control. Each switch position may also be controlled by the computer. The model train system is itself a distributed sensor system capable of representing a broad class of such systems. A substantial number of train sensors are distributed around the train layout and several input "throttles" allow users to enter and control the desired train activity. The train sensors provide event data (interrupt, state and location) and the throttles provide analog input data (via A/D converter).

Any system may have its sensors divided into two categories; those that provide event information and those that provide analog data. Each of these categories may be easily mapped onto the train and throttle sensors. As each sensor in the train system provides identification information along with any data sent to the computer (observer), it is clearly possible to associate separate actions with each sensor represented by the system. The simulated action would typically be either printed on a computer terminal or recorded on the disk system. Real actions that cause the train to move around the layout underlie these simulated actions.

The sensor interrupts are buffered one level in hardware which allows the computer sufficient time to effect various queuing algorithms for concurrent events. Moreover, the software environment for the system embraces multiple real time tasks. This in turn facilitates the simulation of real time operations.

The computer controlled train system is but one of the facilities in the Real Time Computer Applications Laboratory. Other facilities involve acquisition and computer control applications. It is also possible to represent multilevel observer systems by interconnecting the computers.

1.3 Real Time Computer Applications Laboratory The Real Time Computer Applications Laboratory was established in 1976 to permit study and instruction on a wide variety of applications in which the computer is but one component. Distributed sensor problems clearly fall in this category and the broader facilities of the laboratory are pertinent to their study.

The development of the laboratory involved the creation of an environment for learning that encompassed both hardware and software considerations. The hardware present in the laboratory includes typical physical hardware to represent a broad set of real situations which could be encountered, and a set of analog computers which may be used to simulate a broad variety of systems dynamics.

The software provided for the laboratory is critical to the successful utilization of the hardware. It includes both a real time operating system and a higher level language for real time operation. The operating system addresses typical problems such as:

1. Concurrent operations (e.g., concurrent I/O and CPU utilization)

2. event timing control for real time operations

3. recognition of asynchronous events

4. multiple tasks including the handling of priorities

5. interprocessor communication

In addition a specially designed higher level language addresses many of these same problems: multiple tasks, priorities, overlapped I/O, and specialized data types.

The laboratory context of the computer controlled train system is important to its full utilization. Accordingly the facilities of the laboratory will be discussed in some detail.

1.4 Organization of Report

The remainder of this report will discuss in detail the facilities of the Real Time Computer Applications Laboratory and its applications to various problems. The next chapter will contain a brief discussion on the development of the laboratory and an overview of the system facilities. The third chapter will contain a logical view of the operation of the facilities. Chapters 4 and 5 describe both the hardware and software support of the laboratory in some detail. Chapters 6 and 7 will discuss the application of the facilities. Chapter 6 will concentrate on the instructional use of the facility and Chapter 7 will consider other applications (such as distributed sensors). Finally the several appendices contain various operating manuals for the facilities and hardware circuits utilized.

2. OVERVIEW OF FACILITIES

2.1 Laboratory Development

In 1975 the College of Engineering at The University of Michigan recognized that an educational void existed in a certain aspect of computer and computer control activities, namely the use of computers in real time applications. As a result, a proposal was funded jointly with the National Science Foundation to acquire equipment to allow studies in these areas. The present real time computer applications laboratory is the result of that effort.

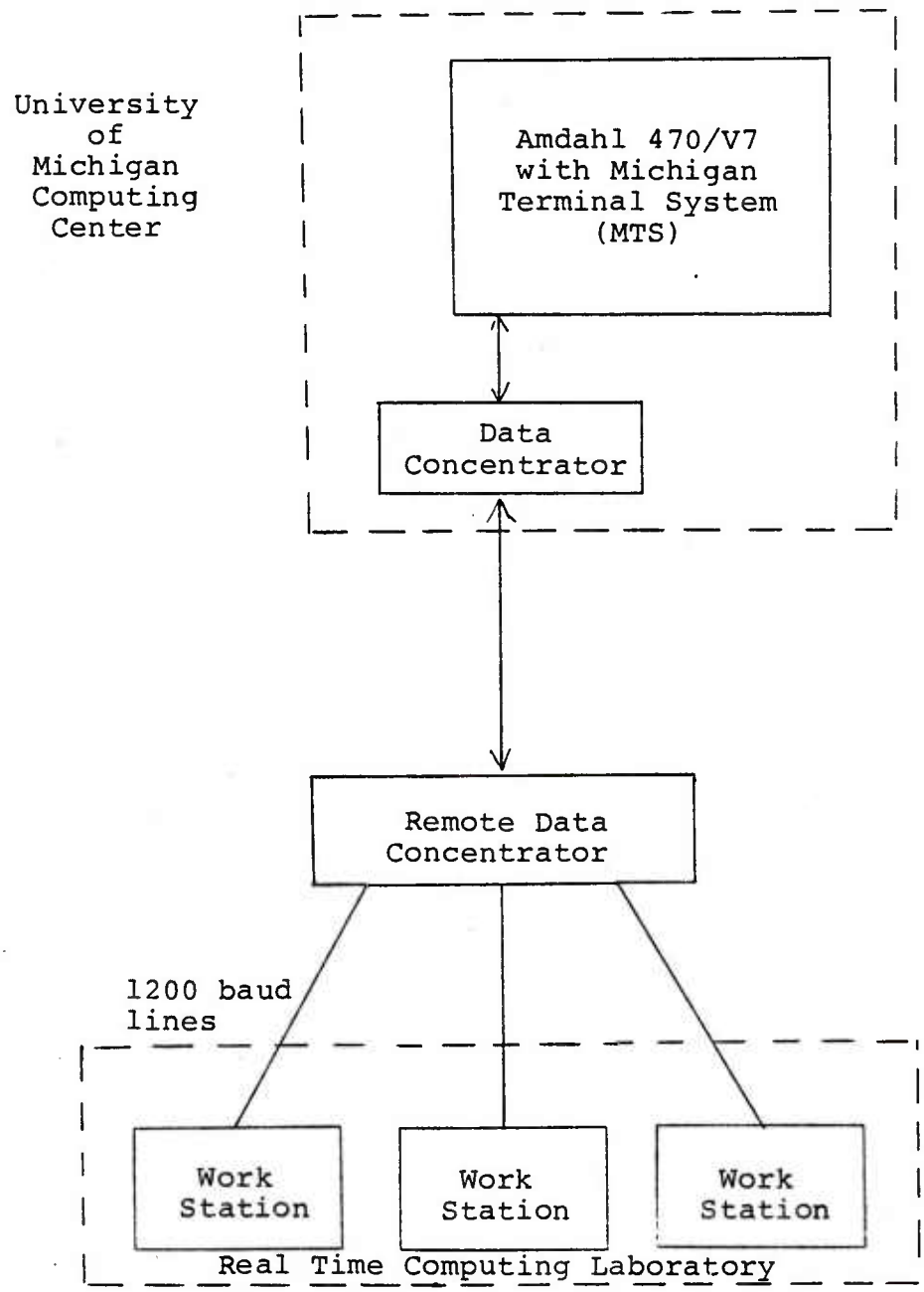
The first equipment was received in January, 1976. The facility was keyed around the newly introduced LSI-11 microcomputer from Digital Equipment Corporation. At that time DEC had very little software support available for the LSI-11 and none suitable for the real time applications envisioned for the laboratory. As a result it was decided that it would be both instructive and useful to develop the major software facilities needed for the laboratories in-house. We felt that we could achieve the needed software support in roughly the same time frame as the vendor and that in the interim we would have prototypes available for use. Moreover it was felt that the experience gained by these developments would have academic merit.

A variety of sources were used for the labor to develop the laboratory. A large number of independent student study projects were carried out to develop the specific aspects of the facility. An advanced class project in compiler writing was used to develop the basics of the higher level language. A small amount of paid assistance was used along with construction help from departmental technicians. The principal direction for the activity was provided by Professor Richard Volz. The development effort included nearly all of the hardware in the laboratory, the floppy disk controllers, A to D and D to A converter controllers, and the entire computer controlled train system. The software developments included principally a real time operating system for the LSI-11 and the definition and construction of a compiler for a real time higher level language, with a substantial number of utilities (e.g., editor and high level debug package) being developed along the way.

Subsequent sections will give a more detailed description of these facilities.

2.2 Overview of Operating Environment

The operating environment for the Real Time Computing Laboratory is shown in Figure 2. Each real time computing work station is connected via a 1200 baud line to a remote data concentrator. The remote data concentrator is an LSI-11 microprocessor attached directly via a 9600 baud line to a data concentrator located at the University of Michigan Computing Center.



The remote data concentrator multiplexes a number of input lines to the University of Michigan's central computer, an Amdahl 470/V7 computer running the Michigan Terminal Timesharing operating system (MTS).

Thus, the real time computing laboratory operating environment represents a four level computer hierarchy. This arrangement is common in many manufacturing and business applications.

Each real time work station consists of an LSI microcomputer system, a floppy disk drive, a Decwriter terminal, access to an analog computer, and various analog to digital and digital to analog devices. In addition, a line printer and N-gauge railroad are interfaced to one of the systems. These microcomputer systems may be configured for independent operation, for operation in communication with MTS, or for multiprocessor operation among themselves.

The system software configuration is structured to take advantage of facilities available on the University of Michigan control computer. All assemblies, compilations, link editing and text editing of programs are done under MTS. Object programs are then either directly down loaded to the LSI-11 or are transferred across the communication channel to the floppy disk for subsequent execution. A small, local operating system known as OSWIT (Operating System WIth Trains) and a cross compiler known as CRASH (Compiler for Real time Applications SHop) have been developed. Other local system utility programs and libraries provide additional capabilities.

2.3 Overview of Hardware

The hardware associated with the mechanical analog facility in the real time systems laboratory is built around three LSI-11 (Digital Equipment Corp.) microcomputer systems. As shown in Figure 3, each system is connected to a floppy disk drive, a Decwriter terminal (30 char/sec.), and various analog to digital (A/D) and digital to analog (D/A) converters. The system is also connected to the university's central computer (Amdahl 470/V7) through a serial interface and a remote data concentrator which allows data transmission rates of 1200 baud (soon to be increased to 2400 baud).

Each microcomputer system is configured so that it can control independently various analog and digital devices, as well as communicate with the central computer for the purposes of program development, cross-compilation, and down-loading of data files and programs. The microcomputers can also be connected together for disk to disk transfer, simulation of a small hierarchy computing system or to allow experimentation with computer to computer communication. Analog computers, simple servo systems, and the mechanical analog facility are

connected to the system through the A/D and D/A converters or through special interfaces.

The independent connection to the central computing system provides the user with access to features of that system that could not be readily implemented in machines as small as the LSI-11 microcomputers. Programs can be developed, edited and compiled without access to the LSI-11 systems which releases those systems for real-time applications.

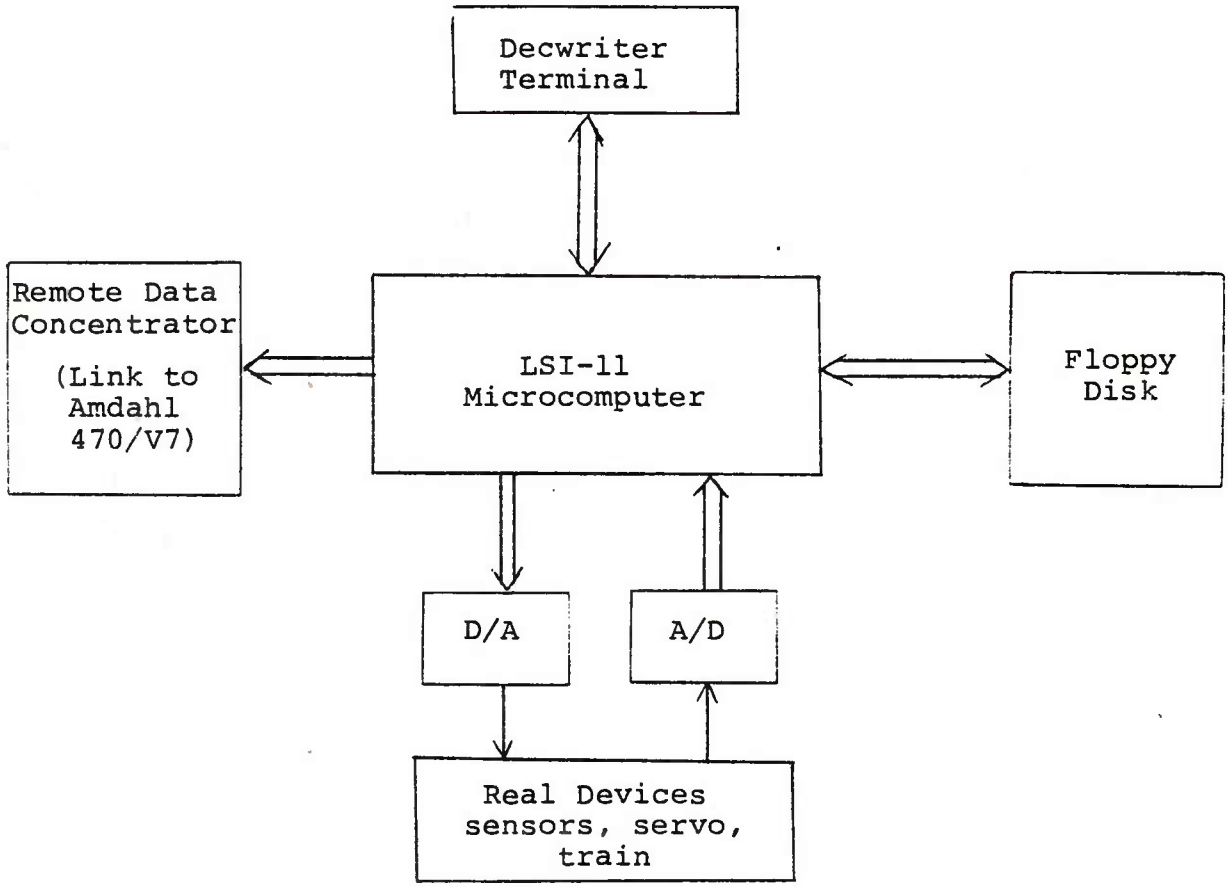


Figure 3. Laboratory Configuration

One of the most important features in the laboratory is the collection of external analog devices that can be monitored or controlled by the microcomputer systems. An analog computer at each station can be used to simulate a wide variety of real time devices with appropriate time and amplitude scaling. Thus the operational characteristics of real world devices can be simulated in conjunction with experiments on real time operation and control algorithms. The control of simple servo systems provides "hands-on" experience with the use of computers and

hardware interfacing that evokes considerable interest in further control studies.

The interface between the disk drive and the computer system was designed and built in conjunction with an early version of the course. This exercise provided students with experience in coupling a high speed, real time device to the system in such a way that it could be serviced according to externally determined timing requirements.

2.4 Overview of Software

An essential part of any computer system is the software available to the users of that system. Certain software tools are necessary for easy and efficient use of the hardware facilities.

The software tools developed for the real-time applications facility are: a small operating system (OSWIT - Operating System WIth Trains); a cross compiler (CRASH - Compiler for a Real-time Applications SHop); and a simple symbolic debugger (RAID - Real-time Applications Interactive Debugger). These tools enable students to write real-time applications programs in a high-level language and debug them symbolically.

The operating system (OSWIT) provides task scheduling, interrupt handling, and I/O control oriented towards real-time applications. A set of utility routines for arithmetic conversion and programs for file maintenance and editing are included in OSWIT. Commands allowing the LSI-11 console to be used as a terminal for communication with MTS have also been provided.

CRASH is a cross compiler that runs under MTS to produce LSI-11 assembly language code as output. This code is then processed through a cross assembler to obtain LSI-11 machine code. The CRASH language is a block structured language similar to IBM's PL/I in its basic control constructs and arithmetic and logical operations. CRASH data types and I/O are specifically designed for real-time data acquisition and control activities. Interrupt handling and task scheduling constructs are available in CRASH to facilitate real-time control.

Symbolic debugging of CRASH programs is possible using RAID. If the debug option is specified, the compiler generates debug tables, and the debug system will be automatically loaded along with the program. RAID enables students to single-step through programs, set break points, and to display and modify variables referenced by name.

3. LOGICAL VIEW OF FACILITIES

3.1 A/D and D/A Converters

There are two A/D and two D/A converters available on each microcomputer system for general purpose application. The specifications and operating details for these 8-bit converters are given in Section 4.1.

3.1.1 Logical Interface

The D/A converter inputs are located at memory location 167772 for D/A 0 and 167723 for D/A 1. The analog output follows the storage of values in these locations by about 1 microsecond.

The A/D converters are controlled by a command status register (CSR) located at location 167770. The bits used in the CSR are shown in Figure 4 below. Any A/D converter with its mode and done bits set will have a conversion started whenever its data register (location 167774) is read. The done bit clears at the start of the conversion and sets upon completion. An interrupt is generated upon completion if the appropriate interrupt enable bit is set. Reading the A/D data register initiates a new conversion. If the mode bit is zero, the A/D will be free running so that it is performing continuous conversions. The value of the latest conversion is obtained by reading the data register. The A/D converter will not interrupt the processor as long as the mode bit is zero.

The A/D converters are normally operated in a free running mode such that they make a conversion, latch the converted value, reset to zero, and start converting 6 usec later. The latched data are displayed continuously as octal values on three 7-segment display modules.

The data are transmitted to the data register by the computer whenever the appropriate control status register (CSR) bit is set high (1). This generates an interrupt in conjunction with an end of conversion (EOC) signal. The data register is read by the CPU and a DATA READ signal returned to the A/D logic network. This DATA READ signal clears the interrupt and resets the A/D converter to zero.

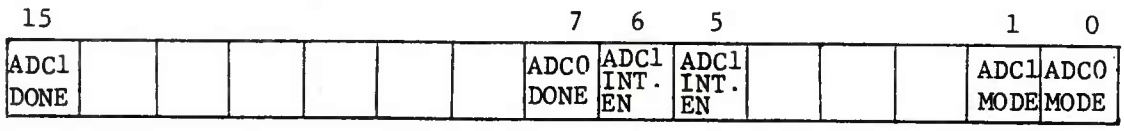


Figure 4. CSR bits for A/D and D/A converters

3.2 Train Facilities

This section deals with the logical aspect of the train facilities, that is, the concepts of operation of the various components and the logical interface between the train and the LSI-11 microcomputer. There are four major components to the train facility.

1. throttle sensors
2. photocell sensors
3. track control
4. switch control

The first two sensors transmit information about the train's location and the commanded speed and direction to the computer. The last two control the train's speed and direction and the position of the track switches.

3.2.1 Throttle Sensors

The throttle sensor is a logical interface used to detect changes in the manual throttle controller. Each train has associated with it a throttle that controls the speed and direction of the train. The throttle controller is simply a potentiometer connected to a 5 volt source. As shown in Figure 5 the output from the potentiometer is connected to an A/D converter. Most programs written to translate the potentiometer settings to speed control treat the center position of the potentiometer as zero speed. Turning the throttle knob in a clockwise direction produces forward motion, while a counterclockwise turn produces backward motion. Each throttle control box also contains a toggle switch used to determine the desired position (straight or turned) of the track switch being approached by the oncoming train. The execution of the toggle switch command is performed by a computer program.

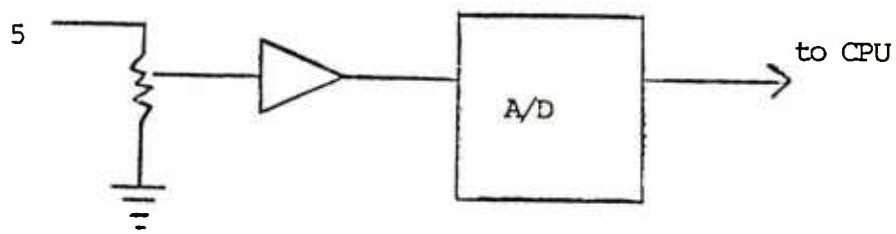


Figure 5. Throttle input

3.2.1.1 Logical Interface

A free running A/D converter is associated with each throttle. The sensor logic continuously monitors both the present and the most recent values from the switch and the throttle. As long as there is no change in either, no action is taken. However, when either the throttle or switch is changed, an interrupt is generated and the new throttle value or switch position is placed on the input to a parallel interface connected to the computer. To avoid flooding the processor with interrupts as the throttle potentiometer is adjusted, the control logic inserts a 100 msec. time delay after each interrupt. During this time no interrupts are passed to the processor.

The bit utilization of the values placed on the input is shown in Figure 6. Only five of the eight available bits from the A/D converter are used. This results in a range of values from 0 to 31 corresponding to the actual 0 to +5 volt range output by the throttle. Since the operating range of the train is -15 to +15 volts, the 0 to 31 range must be mapped into a -15 to +15 range by the computer program.

Each throttle controller has a unique address specified by two bits, which allows a maximum of four throttles. The most recent switch value is contained in bit 13: zero for straight and one for curved.

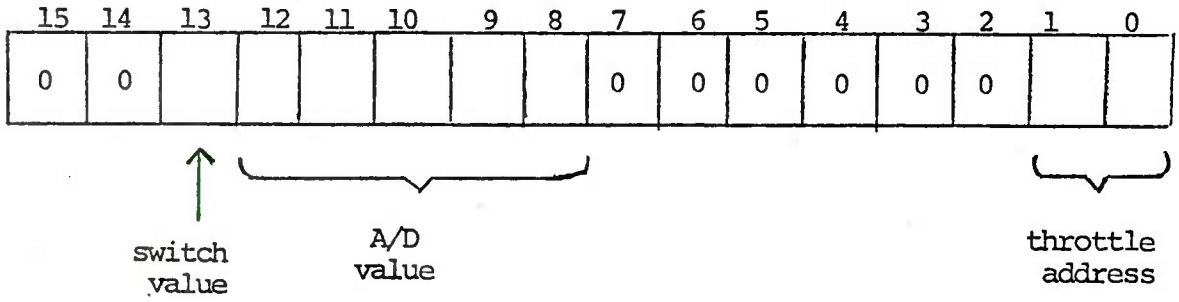


Figure 6. Throttle interface bit utilization

3.2.2 Photocell Sensors

The photocells are used to detect the location of the train on the tracks. Photocells are located at critical points (i.e., those places where the location of the train is essential, such as around the switches and between electrically insulated sections of track). The photocells are adjusted so that the normal ambient room light keeps them on. As a train passes over a photocell, it turns off. The photocells are grouped in pairs to prevent a false indication of an end of train from light between adjacent cars.

Each pair of photocells has a unique address. Whenever one of the photocell pairs changes state (detects a train entering or leaving the area) an interrupt is generated and the photocell address and state are placed on the input to a parallel interface connected to the computer.

The bit utilization for the photocell sensor is shown in Figure 7. Eight bits are reserved for the photocell address, resulting in a maximum of 256 photocells. Our present layout uses only about 64 of these. Bit 8 represents the state of the photocell. A 0 indicates an off or covered condition, a 1 means the photocell is on or uncovered.

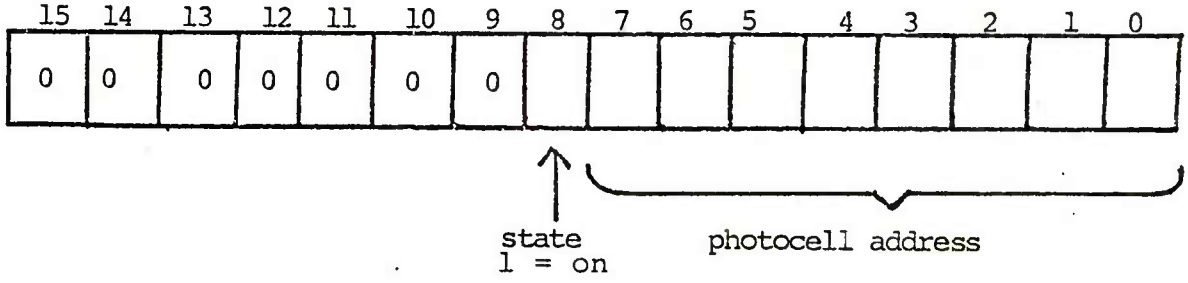


Figure 7. Photocell input word

3.2.3 Track Control

The train layout (see Appendix C) is divided into about 40 electrically separate sections of track. Each section may be separately addressed and has its own D/A converter circuit to supply power to the track. Computer control of train speed and direction is handled by placing an appropriate 16 bit value on the output of a parallel interface connected to the train controller logic network. This 16 bit quantity will supply speed and direction values to a single track address. The logic controller for the train then stores the commanded speed (which is passed on through the D/A converter) and direction polarity for each track section.

A computer control program must address appropriate tracks at appropriate times and send the proper values to the individual track controllers. This process must be accomplished dynamically as the photocell sensors indicate that a train has reached the end of an individual track section. Any simple procedure that applies the same power to all track sections is inadequate for two reasons:

1. It fails to handle loop situations (i.e., a positive to negative short would occur when the train passes over the conjunction).

2. It fails to allow multiple train control.

3.2.3.1 Logical Interface

The track control logic receives a 16 bit integer from the parallel interface each time the user program does an assignment to the interface output. The bit utilization for the track control is illustrated in Figure 8. The tracks are grouped into banks of 64. Bits 12-15 specify one of these banks. Our current configuration supports approximately 40 separate tracks and therefore only bank 0 is implemented. Bits 6-11 are for the address of one track in the selected bank. A six bit buffer associated with each track contains the assigned direction and speed.

When a track is addressed, bits 0-5 are taken from the interface output and placed in the appropriate track buffer. Bits 0-4 contain a value in the range 0-31 which corresponds to a 0 to 24 voltage range. Bit 5 is used to control the polarity applied to the track. A 0 in this bit will cause an engine to move in the direction shown by the arrows on the track layout diagram (Appendix C); a 1 in the direction bit will cause an engine to move in a direction opposite to the direction of the arrows.

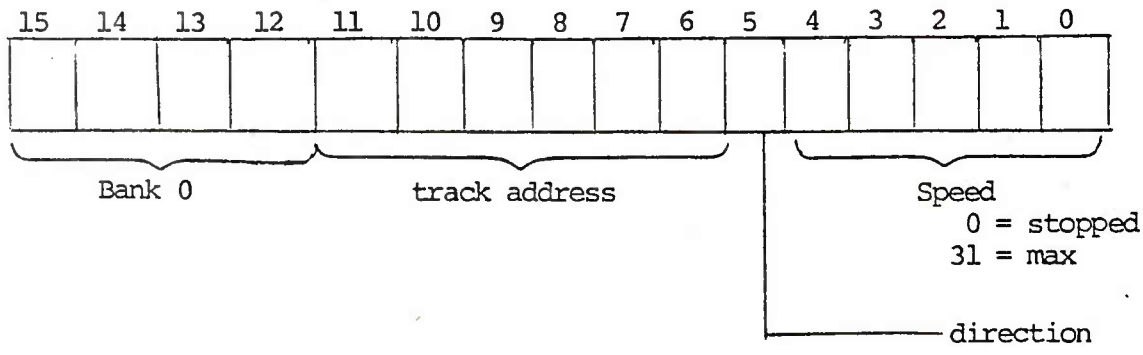


Figure 8. Track Control Word

3.2.4 Switch Control

Approximately 25 track switches are included in the train layout (see Appendix C). These switches allow for two way branching/merging of the tracks. The position of an individual switch depends on the direction of approach of the train. The first case occurs when the train is approaching the switch in a "branching mode". That is, a decision must be made as to which of two directions the train will go. This decision is arbitrary and is made by the user by setting the toggle switch on the throttle control box to either straight or turned. When the computer recognizes that a train is approaching a switch in this "branching mode", it reads the position of the toggle switch and set the track switch accordingly.

The second case of switch control occurs when the train is approaching a switch in a "merging mode". That is, there exists only one direction that the train can continue from the switch. The decision in this instance must be made so as not to derail the train. Upon recognizing that a train is approaching a switch in the "merging mode", the computer program must set the switch accordingly. The switch command set by the user is ignored in this case.

3.2.4.1 Logical Interface

The bit utilization for switch control is shown in Figure 9 The switch control logic is driven by the same 16 bit parallel interface output as the track control. A switch control command is distinguished from a track control command by reference to track bank 15 which is is interpreted as a switch control.

The switches are grouped in sub-banks of 8 each. Bits 8-11

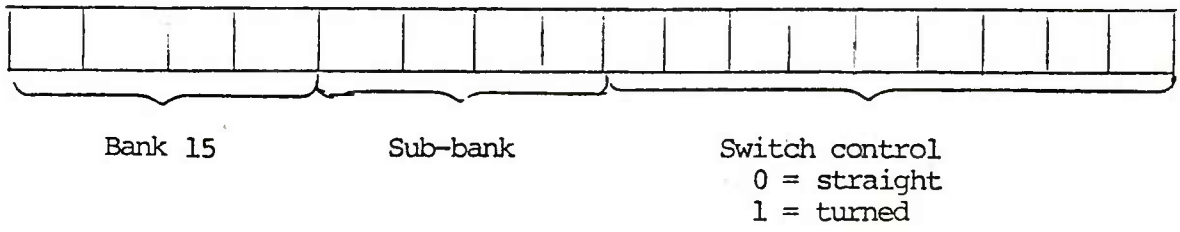


Figure 9. Switch Control Word

are used to address a particular sub-bank. The sub-bank assignments are:

Sub-Bank	Switches
0	0-7
1	8-15
2	16-23
3	24-31

Each of the eight least significant bits (bits 0-7) controls one switch in the assigned sub-bank. A bit value of 0 indicates a straight position, while a 1 indicates turned.

3.3 Software Control of Multiple Trains

One of the principal student assignments using the train setup is to control multiple trains in a transparent manner. That is, the computer is to allow multiple engines to run on the train system; yet it is to appear to the user that the train responds to his control as if he had direct control over each engine in the usual model railroading sense. Moreover, the computer is to arbitrate all contention situations that arise and prevent either derailment and/or collisions.

We will discuss the general problems involved in such control, general approaches to the resolution, and some alternatives which have been used. In implementing this type of throttle driven control there are three critical considerations:

- formulation of a data structure to describe the physical and electrical connections of the train layout
- actions to be taken upon a throttle interrupt,
- actions to be taken upon a photocell interrupt.

3.3.1 Throttle Interrupt

The throttle interrupts are the most straightforward of the three major components to the train control system. An interrupt is generated any time one of the throttle or switch settings is changed. The corresponding actions are relatively simple. The data structure that describes the train layout must also include a structure to store those track sections presently occupied by each train. When a throttle interrupt occurs the control program reads the value inputted by the throttle controller. Since the throttle number (train identification) is included in these data the program then merely looks up the track sections occupied by the train and resets their throttle speed in accordance with the new commanded speed.

Any new position of the command switch is also saved. No action is taken for this switch setting until the train covers a photocell approaching a switch.

Several variations are possible in more elegant control programs. For example, one might implement a speed limit on certain sections of track to control the maximum speed with which a train could negotiate a curve or switch. Likewise the switchyard can be programmed such that trains will not run into the bumper stops at the dead-end of several track spurs.

3.3.2 Photocell Interrupt

The most complex set of actions which must be undertaken occur upon photocell interrupts. Recall that photocell interrupts occur when a photocell is either covered or No action is taken for this switch setting uncovered and that the input information provided by the controller includes the number and state of the interrupting photocell. There are a substantial number of distinct situations that require specific action. Typically the number of the photocell involved in the interrupt will indicate which case is to be performed.

We will assume that the information required by the control program is available somewhere in the data structure without worrying for the moment about how this is done. It is easiest to consider first the actions required by a single engine on a track without considering the complications arising from the possibility of interference with other trains. The variations to account for multiple trains will be considered later.

3.3.2.1 Simple Track Junction

The simplest situation is that occurring when the train is about to pass from one electrical section of track to another, as illustrated in Figure 10. In this and subsequent figures the trains are assumed to be traveling from left to right. The sequence of events and actions is as follows:

1. The train covers photocell A which generates an interrupt to the processor and passes the photocell number and the covered status to the CPU. The control program uses the photocell number as an index to look up the number of the track associated with photocell B. Power is then applied to this track. At the same time the internal data structure is updated to include track N in the list of active track sections occupied by the train.

2. Photocell B is covered which generates an interrupt to the processor. No action is taken.

3. Photocell A is uncovered which generates an interrupt to the processor. No action is taken.

4. Photocell B is uncovered which generates an interrupt to the processor. At this point the train has moved completely off track M and onto track N. Therefore, the control program would remove track M from the lists of track sections occupied by the train and set the power on track M to zero.

Note that events 2 and 3 could occur in either order depending upon the length of the train.

There are several variations which the control program must be able to handle in this case. First of all, the user may reverse the throttle after covering A but before B is uncovered, thus backing the train off track N. This is essentially either step 4 or steps 3 and 4 applied to track M instead of track N and should cause no problem.

A second variation occurs if either track section M or N is

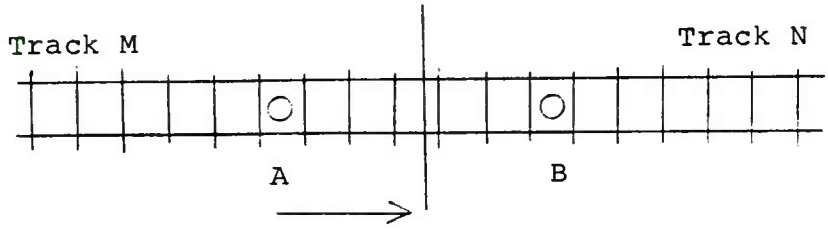


Figure 10. Simple Track Junction

physically short; the train may actually occupy more than two track sections. It is for this reason that one should keep a list of tracks occupied by a given train and their speed controls. In terms of the photocell operations this means that there may be an intervening cover of a photocell requiring a new track allocation before step 4 is reached.

3.3.2.2 Switch Entrance--Variation 1

Assume that the train is entering a switch from the left, as shown in Figure 11. It matters not whether the train enters from position A or A'. The operations to be performed are identical to those in the previous section with one exception. Before powering up track section M the control program must scan its data structure for the position of the switch necessary to avoid derailment. Under photocell A the desired position will be straight and under photocell A' the desired position would be turned. Thus, in this case the switch must be set to avoid derailment as a further action during step 1 above.

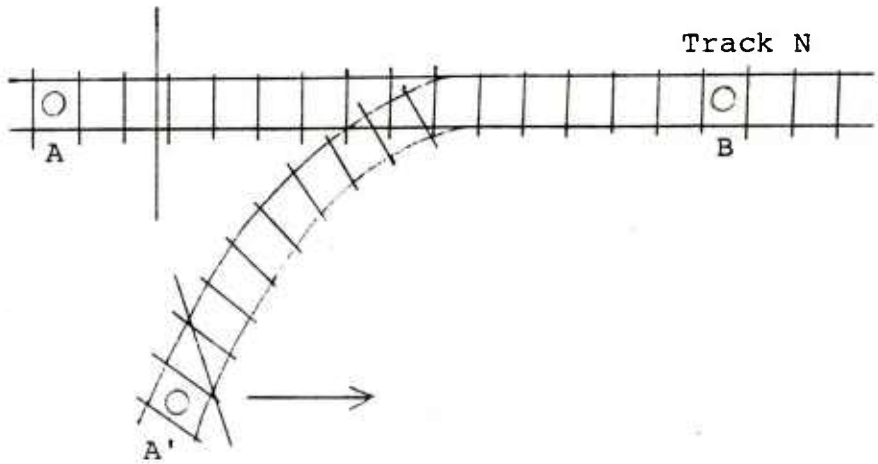


Figure 11. Switch Variation 1

3.3.2.3 Switch Control--Variation 2

Consider a train entering another switch from the left, as shown in Figure 12. The control here is similar to that of the simple track transfer example considered in Section 3.3.2.1. In this case, however, step 1 must be modified to include a determination of the proper switch setting. This is done with the stored switch position command from the last throttle interrupt. The data structure for photocell A must include two track section identifications, that for track N and that for track Q. The control program uses the stored switch position from the last throttle interrupt to determine which track section is to be powered and then issues the appropriate switch setting commands. Otherwise the control is as in Section 3.3.2.1.

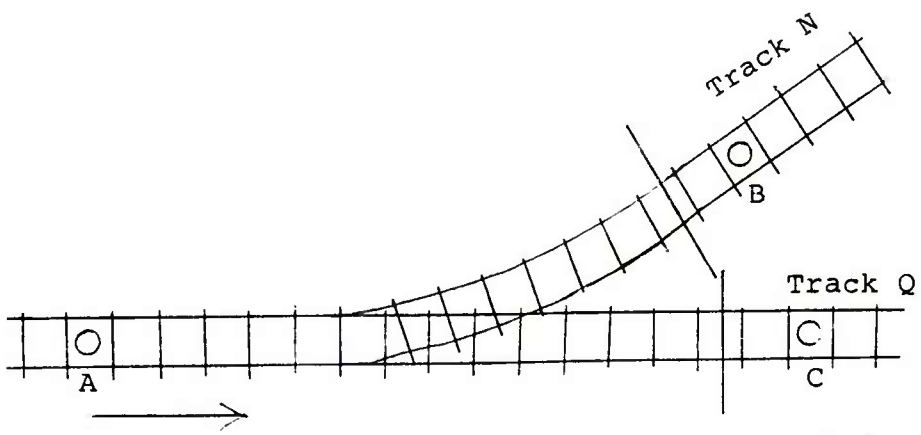


Figure 12. Switch Control--Variation 2

3.3.2.4 Crossover

Figure 13 shows a train crossover. It does not involve any changing of track power but is merely concerned with track occupancy to avoid a collision. Upon covering photocell A the control program must check section track N for occupancy. If section N is occupied and either photocell C or D has been covered (with the train moving in the appropriate direction) it is not safe for a train on section M to proceed. Accordingly, the control program must set the power level on section M to zero to halt the train. Upon throttle reversal (which generates a throttle interrupt to the program) the train may be backed-up.

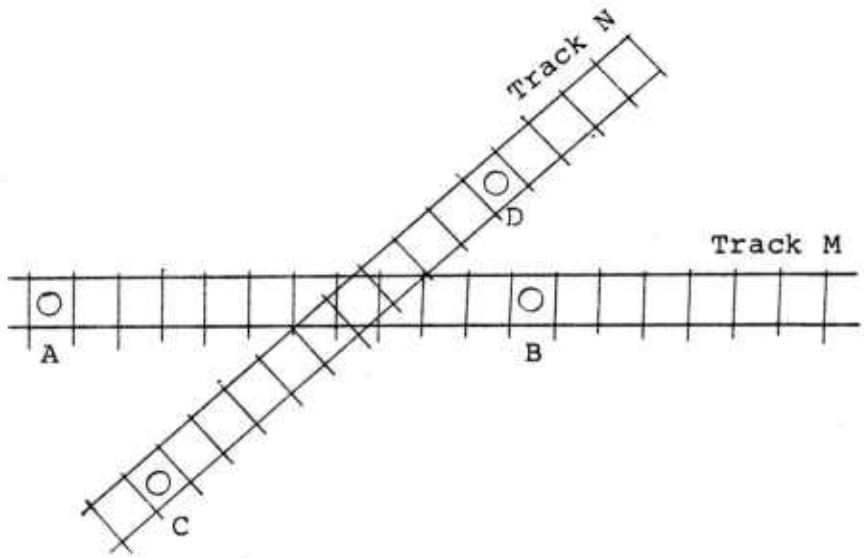


Figure 13. Track Crossover

In the case of multiple trains, when the train has passed over section N (cleared the crossover), power may be reapplied to section M and the stopped train allowed to proceed from photocell A.

3.3.2.5 Loop Control

One of the more complicated issues occurs when a loop is encountered, as in Figure 14. The important thing to notice about this situation is that the inside rail meets the outside rail if one follows it around the loop. Thus, to avoid an electrical short circuit there must be polarity reversal of the applied track power even though there is no reversal in the commanded direction of the train from the throttle. The key

issue in these situations is to identify the presence of a loop, or alternatively, to define a general algorithm to determine polarity of power applied to the track.

In this simple case the existence of a loop is immediately obvious. However, such is not always the case. For example, a close look at the train layout of Appendix C will reveal that there are in fact two loops which occur on the train layout. The second loop involves a number of track sections, not just two. It thus becomes difficult to identify just where in the loop the power should be reversed. In fact there is not necessarily a unique answer in that the loops can be entered from several different points. A more general polarity determination algorithm is needed to handle such problems.

In developing a solution to this problem, consider the movement of the train on a single section of track. Each active track section has a polarity associated with it related to the value of the direction bit in the output word for that section. Each section of track has one end designated with an R and the other end designated with an F, with R and F chosen such that a train will move from R to F if the direction bit for that track is 0 and will move from F to R if the direction bit is 1. The required polarity reversal is illustrated in terms of this notation.

Consider in Figure 14 that a train enters from the left with direction bit set to 0 and the switch is in the turned position. Assume throughout this example that the commanded train direction does not change. The train will proceed through the turned portion of section 1 and approach the R end of section 2. To proceed onto section 2 power must be applied and the direction bit for section 2 set to a 0. When the F end of section 2 is approached section 3 must be powered up with the direction bit also set to 0. When the F end of track 3 is approached, however, the switch position must be set to straight and section 1 powered up with direction bit 1. That is, the polarity must be reversed even though the commanded train direction has not changed.

To develop an algorithm to solve this problem one must consider the possibility of changes in the commanded direction as well. Moreover, the occurrence of a loop may not be at all apparent. The loop may well involve a dozen or more sections of track and be obscured by crossovers and multiple switches. One would therefore like to develop an algorithm that does not depend on loop detection but rather only on the parameters associated with a given track section and the instantaneous state of the commanded direction.

Let us therefore introduce three variables which describe the track orientation, the commanded direction, and the desired direction bit for the track. Let DIR(k) be the direction bit for

track section K which is the variable we are trying to determine. Let EDIR be the variable which represents the desired engine direction. (The particular algorithm by which EDIR is determined from the throttle setting is of no concern here as long as it is a consistent algorithm.)

The first issue of concern is when any new value of DIR(k) has to be determined. Clearly this has to be done whenever the user changes the throttle position and a throttle interrupt is generated. It also, however, must be done when the train is about to pass from one track section to another.

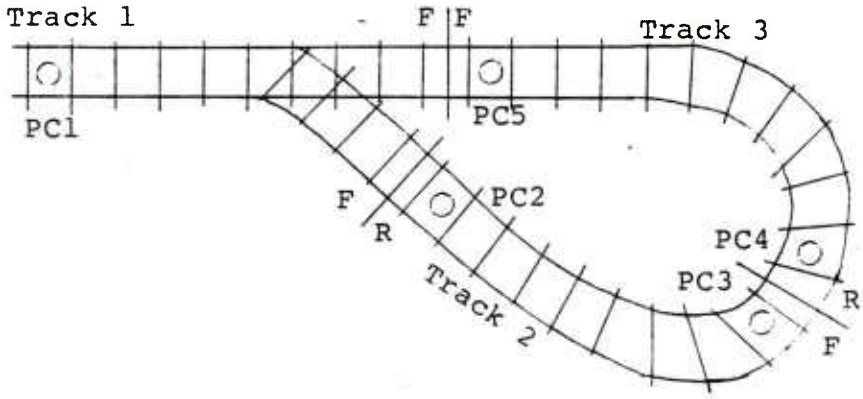


Figure 14. Polarity Reversal in Train Loop

This latter situation is indicated by a photocell interrupt from the photocell just covered. However, not all covered photocell interrupts represent entrance to a new track. For example, in the previous illustration when photocell 2 was covered the train had just entered track 2 and was not about to enter any other track. To detect this situation we associate with each photocell a logical variable END(PC#) according to the following definition.

$$\text{END(PC\#)} = \begin{cases} 0 \text{ if photocell is at R} \\ 1 \text{ if photocell is at F} \end{cases}$$

A train on track section K receiving a covered photocell interrupt from photocell j is about to enter a new track section if the following condition holds:

$$\text{END}(j) \text{ XOR } \text{DIR}(k) = 1$$

The next problem is to determine the polarity, i.e., the variable DIR of the track about to be entered. For this purpose we introduce a new variable, END.NEXT. This parameter is again

indexed by the photocell number. However, as in photocell number 1, the end of the track about to be entered depends upon a switch setting. Accordingly, the parameter END.NEXT must also be indexed by the switch number and the commanded switch position. Thus we define END.NEXT as follows:

$$\text{END.NEXT(PC\#,SW\#,POS)} = \begin{cases} 0 & \text{if next track end R} \\ 1 & \text{if next track end F} \end{cases}$$

Since we know that we are about to enter the next track we can determine its direction bit from this END.NEXT variable. In particular if M is the next section to be entered the direction bit for section M is

$$\text{DIR(M)} = \text{END.NEXT(PC\#,SW\#,POS)}$$

The above algorithm is reasonably straightforward and depends only upon track parameters and the direction bit of the previous track section. It is possible, however, that the user might reverse the throttle while on a section of track. Then the direction bit on the track section must also be reversed. This is easily remembered by recording the commanded engine direction upon entry to a track and simply comparing the commanded engine direction with the commanded direction when the track was entered at each throttle interrupt. More precisely let EDIR.E(k) be the commanded engine direction when track k was entered, and let DIR.E(k) be the assigned value of the direction bit when the track was entered, i.e., the variable DIR.E is obtained directly from END.NEXT of the previous track. Then the direction bit at any throttle interrupt is calculated as:

$$\text{DIR(k)} = \text{DIR.E(k)} \ \text{XOR} \ (\text{EDIR.E(k)} \ \text{XOR} \ \text{EDIR})$$

We thus have two equations to calculate the appropriate direction bit, one when a track is entered and a second when a throttle interrupt occurs. In addition we have an expression telling us when a covered photocell interrupt indicates that the train is about to enter a new track section. With these expressions it is possible to handle the loop problem without any complicated loop detection algorithms.

3.3.2.6 Multiple Train Considerations

It is necessary to add only a few new checks to the control program to include multiple trains. When a photocell at the end of a track section is covered and it is necessary to power up a new section (via any of the schemes described above) the control program must now check the next track section to see if it is already occupied, i.e., the data structures for the layout must include some way of identifying occupied sections. If the section is occupied, the train about to enter that track section must be stopped in its present location and not allowed to proceed. Otherwise, the probability of a collision is high.

When a photocell is uncovered and it is desired to de-power a section of track it is also necessary to check to see if there is a train waiting to enter the track about to be de-powered. If this is the case that track section must be reallocated to the waiting train which is then restarted after release of the track by the first train.

It is also possible for a train stopped and waiting for a track to clear to be reversed without the awaited forward track clearance. Thus, an additional check must be added to the throttle control to provide for a possible train restart by reversal.

With these few additions to the control program it is in principle possible to handle multiple trains. There is, however, one other practical consideration which should be taken into account. This consideration is really only an implementation consideration and does not change the operating principles described above. Multiple trains can generate multiple photocell interrupts at closely spaced points in time. It has been observed that this sometimes causes a failure in the train control program. The problem can be alleviated by not taking direct action upon the receipt of a photocell interrupt but rather storing the photocell interrupt information on a queue. A secondary task is then started to empty the queue. The interrupt driven storage task can then be quite short which minimizes the chance that an interrupt will be lost due to two trains hitting photocells at nearly the same point in time.

3.3.3 Data Structures

By now it is evident that the major key to the control of the train system by computer lies in the selection of data structures both to represent the physical and electrical layout of the track and to keep track of the operating state of the system. Versions of the train control program have been written with different data structures. Rather than state a specific structure, the range of structures will be described.

One can delineate a finite number of circumstances which can arise in terms of covering or uncovering photocells and the direction of train travel at that time. At one extreme users have coded all of this information by hand and stored it with the data structure describing the track layout. Generally, this data structure is indexed around the photocell number. When a photocell interrupt occurs the proper action is determined merely by a table lookup operation. This approach leads to a relatively large data structure and a relatively short and simple program.

The other extreme is to highly encode the data describing the track and write a somewhat more sophisticated program to calculate (as in the loop operation described above) the

appropriate actions.

The most suitable mode of operation with either approach seems to be to store the data that describes the track layout in a separate disk file and to dynamically load it into the program data structures as an initialization phase before the program takes control of the train. This allows the track layout structure to be changed or corrected without recompiling the control program.

4. PHYSICAL DESCRIPTION OF HARDWARE FACILITIES

The analog facilities are divided into three major categories for the purpose of this discussion. The data conversion facilities will be discussed first, then the train facilities, and lastly the other facilities.

4.1 A/D and D/A Conversion Facilities

Two 8 bit A/D binary counter converters (Datel, Model ADC-89A) are associated with each system. These converters are operated as monopolar devices with an input range of 0-10 volts and a 200 microsecond maximum conversion time. They can be operated either in free-running or interrupt mode under program control. The analog input signal to each A/D converter is limited to 10 volts maximum by a zener diode-operational amplifier buffer stage on each input. This signal can be read with an analog voltmeter and its digitized version can be viewed on three octal digit 7 segment display modules.

At the time of selection (about three years ago) these converters were relatively low cost ($70.00) with reasonable temperature stability and conversion speed. Since high precision rapid conversion is not important in the instructional context of the laboratory, these 8 bit converters are more than adequate.

Two 8 bit D/A converters (Datel, Model 198B) are available on each system. These converters are also operated as monopolar devices with an output range of 0-10 volts and 20 microseconds settling time. The offset and gain of each D/A converter can be set through adjustment of two potentiometers. The analog output can be displayed on an analog panel meter and the digital input can be read on the 7 segment display modules. These general purpose converters are also relatively low cost ($29.00) with reasonable temperature stability and conversion speed.

The four converters, the three digit digital display, the analog voltmeter, and the two switches that determine the signals to be read are mounted on a single panel that serves as a preplate for each computer setup. The analog input and output signals are connected to banana jacks mounted on the preplate.

4.2 Train Facilities

4.2.1 Sensor System

4.2.1.1 Throttle Sensor

This section gives a detailed description of the hardware for the electronic throttle (see ELECTRONIC THROTTLE BOARD circuit in Appendix D). A logical description of the throttle can be found in section 3.2. In normal operation the train speed and direction are controlled by a computer program. The program receives data from an electronic throttle control box operated by the user. Each train has its own throttle control box.

The throttle consists of a 500 ohm potentiometer connected to a +5 volt source. The voltage across this resistance is fed into an operational amplifier (MC 1458) and the output from the op amp is in turn inputted to an A/D converter (DATEL, ADC-89A). Only the "middle" five bits of the eight available from the A/D are used, as greater precision is not necessary for train control These five bits are used as input to a six bit D-type latch (SN 74174). The remaining latch bit is used for the commanded position of the track switches. Two poles of the position toggle switch are connected to the preset and clear pins of a D-type positive edge triggered flip-flop (SN 7474) to provide this position signal.

If the controls on the throttle box are not changed, no new information concerning the train's speed or direction is available. Only when a change occurs is there reason for the computer to be interrupted by the throttle control logic. Thus, the purpose of the six bit latch is to hold the past value from the throttle. The six bits at the latch input are compared to the six bits at the latch output by two four bit comparators (SN 7485). If the two quantities are equal the clock input to the latch is kept low (inhibited) and the latch is not triggered. However, if they are not equal, the low signal from the comparator is used to trigger the latch and send an interrupt to the computer (via the NOR gate IC4--see discussion below).

A problem with generating interrupts in this manner is that an interrupt will be sent to the computer for every incremental change in the throttle potentiometer. Since the rate at which these interrupts can be generated is governed by the speed of the A/D converter and the 4 bit comparator, the processor could become flooded with interrupts. To circumvent this problem two one-shots (SN 74123) provide a 100 ms delay between interrupts.

The end-of-conversion signal from the A/D converter is inputted to the first one-shot. The normally high output of the one shot feeds into one input of a two input NOR gate (SN 7402). The other input comes from the comparators. Now, when conversion is complete, the one-shot will go low. This allows the output

from the comparator, via the NOR gate, to control the triggering of the latch. If there has been a change, the new value is latched, and an interrupt sent to the processor. The conversion process is restarted by a second one-shot triggered from the first one-shot.

The process of generating an interrupt and placing the appropriate information on the parallel interface to the computer starts with the output from the NOR gate described above. The pulse from this NOR gate is used as the clock input to a D flip-flop (SN 7474), whose D input is always held high. The complement (low level) output of the flip-flop is passed through a bus buffer gate (DM 8093). The inverted output of the bus buffer is the actual interrupt signal. It is sent to the LSI-11 as a throttle interrupt request. Simultaneously, the address of the interrupting throttle and corresponding A/D values must be placed on the data lines to the computer.

As mentioned in the Logical Description section, a maximum of four concurrently running trains is allowed. Each train is distinguished by the throttle that controls it and each throttle has a unique address between zero and three. These throttle addresses are generated by a counter (SN 74161). The two line output of the counter will contain the address (0-3) of the interrupting throttle. The two outputs of the counter are fed into a two to four decoder (SN 74155). Basically, each decoder output selects one throttle and (in its turn) opens the bus buffer gates to pass any interrupts which may be present for that throttle to the processor.

The clock input to the counter is provided by a continuously running oscillator (NE 556). The output from the oscillator is used as one input to a two input NAND gate (SN 7400). The other NAND input comes from the bus buffer line. When a low signal (interrupt) is present on the bus buffer line, the oscillator is effectively disabled (via the NAND) from the clock input of the counter. This action suspends the counting sequence.

The complement (low level) outputs of the decoder have three important functions. First, they are used to control the bus buffer gates that select the interrupting throttle. Each of the four lines from the decoder corresponds to a bus buffer gate (which in turn corresponds to one of the throttles). When a decoder line goes low, the corresponding buffer gate is selected. If the bus buffer input is also low, the signal will be placed on the bus buffer line and the counting sequence will be suspended. Thus the output of the counter will contain the appropriate throttle address.

Second, each decoder output line corresponds to one of the buffers containing the six bits of data from the throttle control box. Again, when the decoder line goes low it selects

one of these buffers. This places the data on the parallel interface to the computer

Finally, the decoder lines select one of the four NOR gates (SN 7402) connected to the clear input of the D flip-flops associated with each throttle. After the computer has processed the interrupt it sends a DATA TRANS pulse when the information on the data lines has been transmitted to the computer. This pulse is also sent to the four NOR gates just mentioned. The output of the selected NOR is inverted and used to clear the interrupting D flip-flop. Once cleared, the counter will resume and the entire sequence can begin again.

4.2.1.2 Photocell Operation

Overview

The PHOTOCELL CONTROL BOARD is divided into three levels of circuitry. At the lowest level is the PHOTOCELL SLAVE BOARD (see Appendix D). Photocell sensors are placed around most of the switches on the track. These sensors are grouped in pairs with each pair having its own unique address. When an engine covers a photocell, the sensor is turned off. When uncovered, the photocell is turned on. Each SLAVE BOARD consists of a photocell pair connected to a PHOTOCELL MASTER BOARD. The MASTER BOARD prevents any false indication of a change in state and sends the state of the photocell to the PC LOGIC CIRCUIT. If a change of state in one of the photocells occurs, the LOGIC CIRCUIT forwards a signal to the next higher level. Page 1 of the CONTROL BOARD schematic diagrams shows the hardware used to monitor each LOGIC CIRCUIT in order to detect any change in state of a photocell. The CONTROL BOARD then reports the change to the INTERRUPT CONTROLLER (see COMPUTER INTERFACE BOARD). If the device's interrupt system is enabled, an interrupt to the processor is generated through the vector address associated with interrupt A.

Each time a photocell generates an interrupt, its address and state are input to the parallel interface and held until the data is read by the central processor. After the CPU reads the data bus lines from the device, it generates a DATA TRANS to indicate that it has completed the data transfer.

Photocell Address Generation

The hardware necessary to generate the photocell address and state is shown on page 1 of the PC CONTROL BOARD diagrams. The eight bit photocell address is generated by using two four bit binary counters (SN 74161). The output of an oscillator (NE 556) and the inverted PCINT signal (which is generated from the PC LOGIC CIRCUIT) are used to clock the two counters. The eight bits from the counters are connected directly to the input bus lines at the parallel interface on the computer.

Each count corresponds to one photocell pair. The counter outputs are also connected to a set of decoders (described below). With each count, one photocell pair is selected and interrogated for a change in state. The four most significant bits of the photocell address are decoded (SN 74154) to select one of the five banks of photocells. Each bank consists of four PC LOGIC CIRCUITS. Each PC LOGIC CIRCUIT monitors four different photocells (one at a time) on the track in order to determine if one has changed state.

The LAST+1 PC ENABLE line is used to clear the two four bit counters. This LAST+1 PC ENABLE line is the sixth output line from the decoder (SN 74154) used to select the bank. After all the photocells in each of the five banks have been monitored this sixth output line from the decoder will be activated low, which clears and restart the counters from zero.

The least four significant bits of the photocell address are passed through a second 4/16 decoder (SN 74154). The outputs from this decoder are the select lines used to monitor one of the sixteen photocells that are in a bank. These select lines are inputs to the PC LOGIC CIRCUITS.

Change of State Detection

Page 2 of the PC CONTROL BOARD diagrams shows a PC LOGIC CIRCUIT. The inverted D0, D1, D2, and D3 lines are the output select lines from the decoders. mentioned earlier. One of these lines is selected at a time and activated low to monitor a photocell. Only information pertaining to this particular photocell can be passed to another level because of the bus buffer gates (SN 74125). A strobe pulse along with the select line are used as true inputs to a NOR gate (SN7402) to clock a D flip-flop which has stored the last recorded state of the photocell. This strobe pulse is generated by the INTERRUPT CONTROLLER of the COMPUTER INTERFACE BOARD upon receiving a DATA TRANS pulse from the computer. PC1, PC2, PC3 and PC4 are outputs from the MASTER BOARD and contain the current state of the photocells.

The object of the LOGIC CIRCUIT is to determine whether the current state of the photocell is different from that stored in the D flip-flop. This can be done with the use of an XOR (exclusive OR) gate (SN 7486) whose inputs are the current photocell state and the inverted output of the D flip-flop. When the D flip-flop is clocked and a change of state has occurred, the output of the XOR gate will be LOW. The output of the XOR gate is connected to the HCLK line. IF HCLK remains high (i.e., no change in state) the binary counters are incremented by 1 and the next photocell monitored. When HCLK goes low the counters are disabled and an interrupt generated. The photocell address and new state are then sent to the CPU. Notice that when the photocell changes state, the inverted output of the D flip-flop

will contain the new state of the photocell. The inverted output is connected to the DATAOT line which transfers the photocell's new state to a higher level.

Light Detection Circuitry

The lowest level of hardware with respect to the PHOTOCELL CONTROL BOARD consists of the MASTER and SLAVE BOARDS (page 3 of the PC CONTROL BOARD). When light hits the photocell pair of a SLAVE circuit, the sensors act like a battery and generate a signal to the MASTER. When an engine covers the photocell denying it light, a very high impedance prevents any current from flowing on to the MASTER.

The MASTER circuit must be able to send a correct high or low signal to the LOGIC CIRCUIT depending on whether the photocell is on or off. The one-shot is included in the circuit to ensure no waviness in the signal and to prevent a false indication of an end of train caused by light between cars when multiple cars are used.

The output of the amplifier (MC3302) is tied to one input of an AND gate (SN 7408) whose output is also low, sending a low signal to the LOGIC CIRCUIT. The output of the amplifier is also tied to the clock of a positive-edge triggered one-shot (SN 74123) whose negative output is normally high. When an engine covers a photocell, the output of the amplifier is low. The one-shot ensures that a change does not take place for 45 ms. If a false uncover occurs (due to an analog circuit voltage in the undefined region for logic circuit), the one-shot triggers, making Q low; this keeps the output of IC4 low. When the photocell is uncovered by the train, the output of the amplifier is high. The low to high transition causes the one-shot to trigger a low pulse for 45 milliseconds (i.e., a delay is inserted). This approach is used to make sure that the light between cars, if multiple cars are used, would not affect the state of the photocell. Once the one-shot terminates, the output of the AND gate (SN 7408) will be high, sending the correct high signal to the LOGIC CIRCUIT.

Throttle-Photocell Interrupt Management

The INTERRUPT CONTROLLER of the COMPUTER INTERFACE BOARD is simply a two-level priority circuit between throttle and photocell interrupts. The PC INIT signal is generated by the computer upon power up and is used as a power clear to reset all of the D flip-flops in the LOGIC CIRCUITS.

4.2.2 Control System

4.2.2.1 Track Speed Controller

The track control word is organized as shown in Figure 8. The track speed controller selects one of 64 tracks from a six bit track address, (all tracks are on Bank 0). The speed is a five bit number (yielding possible speeds 0-31) and a direction bit.

Track selection occurs as follows. (See TRACK SPEED CONTROLLER 1 of 3 in Appendix D.) Bits 10-11 of the track control word are input to a 1-of-4 demultiplexer (SN74155). The four outputs are used to enable one of the four 1-of-16 demultiplexers (SN74154) that are fed with bits 6-9. The result is a 1-of-64 demultiplexer. (Tracks are numbered 0 through 63.)

Two types of speed control are implemented. The first is the traditional analog dc voltage control used on tracks 0 through 15 (0-17 octal). The second, used on tracks 16 through 31 (20-37 octal), is under digital control: the track is pulsed with a high voltage pulse train, with duty cycle proportional to the desired speed.

The analog controller (see TRACK SPEED CONTROLLER 2 of 3 in Appendix D) works as follows. The incoming speed (5 speed bits + 1 direction bit) is latched. The most significant four bits of the speed are held in a 4-bit latch (SN7475). The least significant bit and the direction bit are held in a dual D-type flip-flop (SN7474). The five speed bits are fed to a home built weighted-resistor digital-to-analog converter. The resistor ladder is composed of five separate resistors. Some attempt was made to match the values when the boards were built (nominal values are 2, 4, 8.2, 16, and 33 K ohms). The operational amplifier used in the D/A is a MC1458. The output from the op amp drives a simple two transistor amplifier (a pair of 2N3053's). This yields a maximum possible voltage of 25 volts. This is directed to the track through a double-pole, double-throw relay (PD RI0-E1-Y4-V52). The direction bit is inverted with a NAND gate (SN 7401) and used to drive a transistor amplifier (a single 2N3055). The transistor provides the power to switch the direction relay (the NAND gate was used as an inverter only because of availability).

The digital controller (see TRACK SPEED CONTROLLER 3 of 3 in Appendix D) differs functionally from the analog in its resolution. All five speed bits + the direction bit are latched in a hex D-type flip-flop (SN74174). The most significant four bits are used to set a four bit presetable binary counter (SN74161). The least significant bit is not used.

The counter is used to pulse the track with a 25 volt pulse. As the duty cycle is increased, the speed increases. A

speed of 0 never pulses the track (duty cycle=0). A speed of 2 sends a 1 to the counter (since the least significant bit is not used) and yields a duty cycle of 1/16=.0625. A speed of 24 (counter input=12) gives a duty cycle of 12/16=.75. Finally, a speed of 31 yields a unity duty cycle. This is accomplished by loading the counter with the upper four bits of the speed and letting it count from there to 15. Power to the track is inhibited while it is counting. During the remaining (up to 16) counts, power is applied to the tracks. One of the counter outputs, TC, becomes high only when the counter reaches 15. TC is inverted and fed back into the CEP input. CEP allows the counter to count only when it is high.

An oscillator (described below) provides clock pulses for both the counter and a load function (once every 16 clock pulses). The load pulse reloads the counter with an initial count and resets TC low. The net effect is to let the counter count from the speed/2 to 15 and stops for the remainder of 16 counts. TC is reinverted and used to drive a dual transistor amplifier exactly as in the analog case. The direction bit is inverted and controls the relay in the same manner. (Note: all inverters on this board are SN7406.)

The oscillator for the digital speed controller consists of a NE556 oscillator and a SN7493 divide-by-16 counter. The oscillator provides the CLOCK input to the SN74161 counter on the speed controller. The SN7493 divides CLOCK by 16 to provide the LOAD pulses for the counter.

While in principle the two types of controllers merely provide two different mechanisms of providing a track voltage proportional to the commanded speed, some pratical differences have been found. Specifically, the output amplifier stage on the analog version of the track controller has been found to be quite nonlinear. The table in Table 1 shows this difference. The operational effects, however, are not as drastic as would seem from this table. There are several reasons for this. First, static friction keeps the train from operating at much below 4 volts. Secondly, there is no printed calibration on the hand held throttle; one only has the "feel" for its operation. Thirdly, the usual operating range is at the higher speeds at which the difference is less. Small speed changes can sometimes be noticed as the train moves from one track section to another, but this will occur even between analog tracks as low precision resistors were used (because they were available and cheap) in the weighted resistor D/A converter.

4.2.2.2 Switch Controller

The switch controller is activated through bank 15. The switches are divided into 16 subbanks of eight switches each. After the programmer selects a subbank the switch controller sets each of the eight switches on the subbank ONE AT A TIME; it does not attempt to apply power to eight switches simultaneously. If all switches do not set properly within a short time, an audible alarm sounds. The switch control word format is shown in Figure 9.

TRAIN TRACK
SPEED-VOLTAGE RELATIONSHIP

Track 0-17 Speed (Octal)	Voltage on track (Decimal)	Track 20-46 Speed (Octal)	Voltage on track (Decimal)
1	1.2	0	0
16	1.8	2	1.1
20	4.0	10	4.4
23	8.2	16	7.8
24	9.5	20	9.0
25	11.0	22	10.0
26	12.0	24	11.0
27	13.5	30	12.0
30	15	32	13.0
31	16	34	15.0
32	17	36	18.0
33	19	40	0.0
34	20	42	-1.6
36	20	56	-7.8
37	20	60	-9.0
40	-1.2	62	-10.0
		64	-11.0
55	-1.2	66	-12.0
56	-1.8	70	-13.0
60	-4.0	72	-14.2
70	-15.0	74	-16.0
77	-20	76	-18.0

TABLE 1

The switch controller is activated initially by selecting bank 15 (bits 12-15) (see COMPUTER INTERFACE BOARD and SWITCH CONTROL circuits in Appendix D) on a 1-of-16 decoder, (SN 74154). The complement of the bank 15 select line is used to enable a second 1-of-16 decoder to select a subbank from bits 8-

11. Only subbanks 0-3 have been implemented (4x16=64 switches.) The subbank select lines are inverted and used to enable exactly one pair of four bit latches (SN 7475). These latches hold the switch settings.

The latched setting bits are XORed with the feedback of the actual switch position. If they differ, the commanded bit is used to set the switch to the desired position.

The feedback for each switch is latched in one half of a dual D-type flip-flop (SN 7474). If the switch position is straight, the flip-flop is set; otherwise it is cleared. The complemented output is used so that a straight switch yields a 1 output and a turned switch yields a 0 output. The XOR is performed with a SN 7486 quad exclusive-or gate. Note: The XOR-ing is performed simultaneously for all eight switches on the subbank.

All eight switches would need switching in the worst case. Since this would require an inordinate amount of power, they are switched sequentially. The outputs from the XOR gates are fed into a 1-of-8 priority encoder (SN 74148). (All inputs to the priority encoder are complemented.) The priority encoder chooses the high order 0 input and encodes its number onto the three output lines. These three lines are immediately decoded again (using an SN 74138 1-of-8 decoder) to select exactly one switch enable line. Since the priority encode inputs are complemented, if the feedback latch (flip-flop) output matches the set bits, the XOR gate puts out a 0 which is eventually encoded by the priority encoder. Inverters (SN 7404) are used to "re-complement" the switch enable lines.

Now consider a single switch (preferably the one that has been selected). The set bit is inverted, and both the original and the inverted signals provide the first inputs to NAND gates (SN 7400). The second input is the select line. The NAND gates select one of two one-shots (SN 74123) which in turn drive SCR's (IR 108 BI), which drive the track switcher, which causes feedback, etc. The purpose of the one-shots is to prevent power from being applied (through the SCR) to a switch controller for more than 150 milliseconds in case a switch gets stuck and doesn't move. A longer switching pulse could burn up the track switches.

Once the feedback matches the set bit, the input to the priority encoder becomes 1 and the encoder encodes the next switch that requires resetting. When all switches on a subbank are properly set, the priority encoder activates its enable out (EO) line. This is connected to the enable in (EI) line of the next subbank and allows the switches there to be set. The propagation of EO is prevented by any switch that does not set. The EO line from subbank 15 (or the last subbank) is connected to an audio oscillator; if EO15 does not come on in a short

time, an audio alarm sounds.

Finally, SPST switches lie between each XOR gate and its priority encoder. These allow a switch to be isolated from the controller if it gets stuck.

4.3 Other Facilities

There are several other hardware facilities that may be used in conjunction with each computer installation. These facilities will be described briefly since they are associated with other applications of the computers.

4.3.1 Servo Systems

The computer system can be used to program the real time control of a servo motor system (Experiment 3 in Appendix F). The hardware facility includes an interface between a computer controlled D/A converter and a servomotor. The interface consists of three operational amplifiers that act to scale and offset the output voltage from the D/A converter to levels compatable with the input stage of the servomotor control unit.

There is also an interface between the output of the reference potentiometer which monitors the output position of the motor shaft and the A/D converter in the feedback loop. This interface consists of two operational amplifiers that scale the potentiometer output voltage to levels compatible with the A/D input to the computer.

The servomotor control has a chopper-stabilized DC amplifier that drives a power amplifier which in turn drives the servomotor. The servo reference potentiometer is a single turn wire wound potentiometer with a total resistance of 50 K ohms.

4.3.2 Analog Computers

An analog computer is available at each computer installation. This computer can be programmed to provide a variety of time varying input signals to the A/D inputs of the computer. The analog computer is also used to provide signal and impedance buffering in servomotor control studies and applications.

The analog computers are Applied Dynamics AD-2 units that contain 16 operational amplifiers and 4 multipliers. The circuits are wired on patch panels that can be inserted for specific applications. These tube based computers are designed to provide output signals in the range of 100 volts.

4.3.3 Data Acquisition System

A simple data acquisition system is available to provide "respiration" signals to the computer (Experiment 2 in Appendix F). This system consists of a thermistor in a cylindrical "breathing" tube. The thermistor is connected as the feedback element in a simple battery driven operational amplifier circuit. The passage of inhaled and expired air through the tube alters the thermistor resistance and thus the output signal from the operational amplifier. An on-off switch and an offset voltage potentiometer are used to control the input signal to the A/D converter.

5. DESCRIPTION OF SOFTWARE SUPPORT FACILITIES

5.1 OSWIT - Operating System WIth Trains

5.1.1 Introduction

The field of digital computers and their applications is perhaps the most dynamic field in engineering at the present time. Driving this change during the past ten years has been the introduction and widespread acceptance of the microcomputer. There are numerous products on the market using microcomputers, and the future is almost limitless. At present, however, software support for these systems lags far behind their older and larger counterparts. The availability of microcomputer operating systems is rather limited. Most present microcomputer operating systems are not really suited to real time applications that are forthcoming for microcomputers. During the next decade it is important that suitable real time operating systems be afforded the developer of microcomputer applications.

OSWIT (Operating System WIth Trains) is an operating system developed at the University of Michigan to meet real time executive system needs for the Digital Equipment Corporation LSI-11 microcomputer. The basic features of the operating system were designed and implemented by Jack Bonn and Ted Kowalski as an independent study project under the direction of Professor Richard A. Volz in the fall of 1975 and winter of 1976. During the fall 1976, Bill Dargel was responsible for the design and implementation of the disk controller. In addition, Kent Hoult developed and implemented the file system while Arnold Vance implemented the A/D and D/A drivers and train interface. In fall of 1977, Houton Aghili completed the design and installation of the MCP protocol between OSWIT and MTS. In fall 1977, Kent Hoult continued the development of OSWIT and the file utilities. Carol Briggs, Mark Haynie and Glen Purdy later modified the I/O structure to allow transmission rates of 2400 baud. Rick Richardson modified OSWIT to support DEC compatible soft sectored floppy disks at other locations within the University.

The basic features of the OSWIT operating system are:

1. A task scheduler which functions with a programmable clock and asynchronous events to start tasks by various methods subject to a specified software priority.

2. A wait structure to allow processing and I/O operations to proceed in parallel.

3. Input/Output device drivers for the console, A/Ds, D/As, floppy disk, paper tape reader, and printer.

4. MCP protocol to allow the microcomputer system to

communicate with the University's central computer system (MTS).

5. A simple command structure modelled after the Michigan Terminal System (MTS).

6. Floppy disk file system.

7. A small set of utility routines to support arithmetic conversions.

8. An absolute loader.

A brief overview of these features will be given here. They are described in greater detail in the OSWIT user's manual, presented in Appendix A.

5.1.2 OSWIT command language

The OSWIT command language provides the mechanism for user communication with OSWIT. This command language is modeled after the Michigan Terminal System command language. This command language permits system control, program control, a debugging monitor, file handling and communication with MTS.

This command language also supports logical unit assignment and pseudo device names similar to those used on MTS. Assignment of the physical devices to logical units may be done when program execution is initiated from the OSWIT command language or from within an executing program.

Pseudo device names are used by OSWIT command language to symbolically refer to physical devices in a manner similar to file references. Pseudo devices names are provided for terminal output and input, paper tape reader, the line printer, the A/D and D/A converters, the train interface and a dummy file or device.

5.1.3 OSWIT file system and utility programs

OSWIT can create, destroy, rename empty, truncate, edit, and catalog disk files. To minimize the operating system memory requirements, these mechanisms are provided by an OSWIT utility program named *FILES11. OSWIT defines a file as a sequence of logical records placed in non-contiguous, 512 byte blocks on the disk. A file cannot exceed 255 blocks.

Filenames are limited to 10 characters or less and may consist of any combination of printable, uppercase characters. Any filename starting with an "*" is designated as a public file and is usually reserved for OSWIT system files and utility programs.

No file protection mechanism is available in OSWIT. The WRITE ENABLE/PROTECT hardware switch is the only protection available.

Other utility programs, such as *EDIT, *FILESNIFF, and *TIME provide additional user support (see Appendix A).

5.1.4 OSWIT support functions

A number of support routines used by the operating system to implement its functions are internal to OSWIT. These include numerical conversions, dynamic buffer management, I/O operations, and task scheduling. As a general principle, all such functions are available to user's programs at the assembly language level via emulator trap instructions (EMTs).

5.1.5 MTS - OSWIT communications

OSWIT uses the MCP protocol {1} to communicate with MTS on an Amdahl 470/V7. Each system is hardwired via a 1200-2400 baud line to a remote data concentrator, which statistically multiplexes each input/output line with other units and communicates with MTS through a hardwired 9600 baud line. This connection is used principally to transfer data and programs between MTS and the local floppy disk, or to use the system as an "intelligent terminal". Source editing can be done locally, transferred to MTS for assembly or compilation, linked and the object file down loaded to be stored and executed on the microcomputer system. Alternatively all development of user programs can be accomplished on MTS with the final object stored and executed locally. In addition, data may be collected and transferred to MTS for greater storage capacity or more thorough analysis.

5.1.6 Real time operations

According to Martin, {2} a real time computer system is one which accepts inputs from one or more sources, acts upon these inputs, and produces corresponding outputs fast enough to effect the source. This definition encompasses a wide variety of systems such as the use of a computer as a data concentrator, as the control element in a feedback loop, as a data logger for some real time process, or as a supervisor for a set of other real time computers.

1 UM Computing Center, "An MTS Communications Protocol (MCP) Proposal", May 1976.

2 Martin, James, Design of Real Time Computer Systems, Prentice Hall.

There are two primary characteristics which distinguish real time application from scientific computations: the need to respond rapidly to the occurrence of events external to the computer, and the need to handle I/O for a potentially large number of external devices in a manner which does not lock up the CPU during the I/O transfer. An example would be to require a computer controlling electric power distribution to suspend normal program operations upon detection of a generator failure and initiate an orderly shutdown procedure for that generator and a redistribution of the load among the remaining generators. The consequences of these characteristics are far reaching.

5.1.6.1 Tasking

First, in order to allow the user to specify the response to external events, he/she must be given some control over interrupt handling. Secondly, since the computer is usually much faster than the devices it controls or responds to, it is common to have a single computer control a number external devices. As a result, one usually has several more or less independent pieces of code known as tasks which are executed at different times. OSWIT provides a mechanism for associating a task with an interrupt or a condition for a given external device. When an interrupt occurs the program currently operating may be suspended and the associated task executed. When this task is completed, its execution is terminated and the original program is resumed.

Associated with notion of task is that of a priority. If two or more tasks are competing for the CPU, there must be some mechanism for deciding which task is to execute. In OSWIT each task is assigned a priority. Once started a task will run to completion unless interrupted by a task with a higher priority. If task A has priority of 10 and is interrupted by task B with a priority of 25, task B will execute until completion unless interrupted by a task with a priority greater than 25. When task B finishes, task A will resume.

OSWIT supports tasks that require synchronous timing. The LSI-11 microcomputer hardware has a programmable real time clock. The user can request OSWIT to set up time intervals in the clock and interrupt the CPU when the interval has passed. This OSWIT facility allows the user to specify that a task is to be executed repeatedly at fixed intervals of time, at a certain time of day or after some interval of time.

5.1.6.2 I/O and interrupt structure

The OSWIT I/O and interrupt structure is generalized and oriented toward real time applications. All I/O operations at the programming level are done through logical unit assignments. Assignment of physical devices to logical units may be done at the time program execution is initiated or dynamically from

within the program. All I/O requests to OSWIT do an immediate return to the calling program after the request is initiated so that processing may be overlapped with I/O. If an I/O operation must be completed before the task can proceed, the task may issue a WAIT request to OSWIT.

OSWIT supports logical record (line), byte, word and character I/O. OSWIT also supports requests for decimal or octal character string to binary word and binary word to decimal or octal character string conversion.

5.2 CRASH - Compiler for Real time Applications SHop

5.2.1 Introduction

The CRASH language and compiler were developed both as a vehicle to assist instruction in the Real Time Computer Applications Laboratory, and as an experimental language incorporating constructs to support real time computer operations.

The CRASH compiler is a cross compiler which executes on the University of Michigan's Amdahl computer. The compiler produces LSI-11 assembly code which is then processed through a cross assembler. The object code is then link edited on the Amdahl and down loaded to the LSI-11.

CRASH is a block structured language and is similar in some respects to IBM's PL/I. A number of unessential PL/I features have been restricted, and data types and I/O statements specifically designed for data acquisition and real time control activities have been added. Task scheduling and interrupt handling constructs to facilitate real time control are also included in CRASH.

These special structures have been added to CRASH not only to facilitate real time programming, but also to permit better program structure and cleaner code. Programs written in the CRASH language may be structured according to a logical plan with separate tasks being utilized for real time activities, thus avoiding a clutter in the middle of the main program which would be necessary using other programming languages. The following section will describe the CRASH language, concentrating on the features designed to accommodate real time applications. A more complete description is available in Appendix B: CRASH User's Manual.

5.2.2 Procedures

Four types of procedures are available to provide for program structure, modularity, and ease in debugging:

EXTERNAL
INTERNAL
MAIN
TASK

The basic unit processed by the CRASH compiler is the EXTERNAL procedure. For each EXTERNAL procedure, a separate assembly program is produced, resulting in separate object modules after cross-assembler processing. Object modules for each external and library procedure in a program are linked together using the linkage editor to produce an absolute load module. This approach has the advantage that the whole program need not be recompiled for a change in one procedure, thus saving time and money in program development.

An INTERNAL procedure is one which is defined within another procedure (either EXTERNAL or INTERNAL). Three nested INTERNAL procedures inside an EXTERNAL procedure are allowed. Procedure calls, however, may be nested to arbitrary depth.

A MAIN procedure is a special type of EXTERNAL procedure, similar to a PL/I MAIN procedure. There must be one (and only one) MAIN procedure in every program. Program execution begins with the first executable statement in the MAIN procedure.

EXTERNAL and INTERNAL procedures may be referenced by calls either as subroutines or as functions. All procedure parameter passing is done by reference.

It is possible to write a recursive procedure by judicious use of AUTOMATIC variables, parameters, and re-entrant code. Return from a procedure to the calling program occurs automatically when the end of the procedure is reached. A RETURN statement will cause immediate return to the calling program. A return value may be passed back to the calling program by using a "RETURN <expression>;" statement. If a procedure is to return a value (as in a function call), the procedure must be defined to be of the data-type(see section 5.2.3 for description of CRASH data-types) which the procedure is to return.

A special type of EXTERNAL procedure, known as a TASK may be defined to facilitate real time operations. A TASK is an externally defined procedure that takes no arguments, and may return no value. TASKs are usually invoked through the use of CRASH scheduling statements. The main difference between TASKs and normal procedures are their ability to be invoked by a scheduling statement, and their ability to run at a different

priority level than the invoking procedures. TASKs, scheduling statements, and priorities are explained in section 5.2.8 (Tasking and Timing).

5.2.3 Data Types and Structures

The basic data types included in CRASH are:

INTEGER
REAL
CHARACTER
BIT
BOOLEAN
ROUTINE
TASK

The INTEGER,REAL,CHARACTER, and BIT variable types in CRASH are roughly the same as in most other languages. All REAL variables are 2 LSI-11 words (32 bits) long. INTEGER variables occupy 1 word (16 bits) as do BIT variables, since packing of BIT variables has not been implemented. CHARACTER variables are of fixed maximum length from 1 to 254 bytes. BIT variables must be declared with a field width specification (1-15), and CHARACTER variable declarations must include a maximum length specification (1-254). BOOLEAN variables are the same as BIT(1) variables.

INTEGER, REAL, CHARACTER, BIT, and BOOLEAN variables may be declared as arrays of up to 62 dimensions. Each dimension is specified as (lower-bound: upper-bound). Negative subscripts are allowed. The lower-bound may be omitted and defaults to zero. If array bounds are specified with variable names rather than constants, storage allocation is postponed until run-time, thus allowing dynamic modification of array size.

Any procedure label referred to in a program must be declared just like any other identifier. The usual data-types apply to procedure label declarations if a return value (of the declared data-type) is expected.

The ROUTINE data-type is used to declare normal procedure labels (i.e. INTERNAL or EXTERNAL procedure identifiers) where no return value is expected.

The TASK data-type is used for declaring identifiers for TASK type procedures. Procedures declared to be of the TASK type also may not send return values.

A number of optional characteristics, known as attributes, may be specified for a variable to provide great flexibility in the use of CRASH variables.

The attributes available in CRASH are:

ANALOG	CLAMP
DISCRETE	MAP
LDN	INTERNAL
DELAY	EXTERNAL
BYTE	AUTOMATIC
WORD	STATIC
SCALE	GLOBAL
OFFSET	INITIAL

The ANALOG attribute is used with certain other attributes to associate additional information with a REAL or INTEGER variable. Typically variables with the ANALOG attribute are associated with an I/O device such as one of the A/D or D/A converters. The declaration of an ANALOG variable may include a logical device number (LDN) through which I/O is to be accomplished. A SCALE factor and an OFFSET to convert external voltages to the units used in the internal representation of a variable may be included if desired. SCALE and OFFSET are applied as follows:

On Input:

```
INTERNAL_RESULT=(EXTERNAL_VALUE/SCALE)-OFFSET
```

On Output:

```
EXTERNAL_RESULT=(INTERNAL_VALUE+OFFSET)*SCALE
```

An optional CLAMP attribute may also be specified for ANALOG variables to prevent wraparound. If the CLAMP attribute (specified as (low_limit, high_limit)) is included, any attempt to output a value outside the specified range will cause the appropriate limit value (low or high) to be output instead. If an 8 bit D/A converter is used with low_limit=0 and high_limit =255 then an attempt to output any value greater than 255 will cause 255 to be output instead, and an attempted output value of -10 will cause an output of 0. Without the CLAMP attribute an attempted output of 260 would result in wraparound with a actual analog output of 4, and an attempt to output -10 would result in an actual output of 245. Since it is usually better to have 255 where 260 was intended rather than 4, the use of the CLAMP attribute with ANALOG variables is recommended when there is any doubt about the range of possible output values.

Variables with the DISCRETE attribute are similar to ANALOG variables in that they may be associated with a particular device by the use of the LDN specification. However, the DISCRETE attribute is valid only for INTEGER variables. DISCRETE variables are intended to be used with status or control registers, where value conditioning is unnecessary but implied I/O port reference is still convenient.

Since some devices are 8 bits wide while others are 16 bits, two attributes, BYTE (8 bits) or WORD (16 bits) are available to indicate the number of bits to be read or written. These attributes may be specified for any ANALOG or DISCRETE variables and default to WORD for all variable types.

One common use of ANALOG variables is in difference equations used to compute new values for some output variable as in a sampled data control system. Typically in such uses, the control formula requires a finite number of past sample values of the ANALOG variable, as well as the current value. Accordingly, a set of past values for each ANALOG variable may be kept automatically by utilizing the DELAY attribute. These past values may be referenced with the form:

```
- ANALOG_VAR@EXPRESSION
```

Where EXPRESSION is either a variable or a constant, or any arithmetic or logical combination of them. The EXPRESSION, called a delay indicator, is converted to an integer before being used. If EXPRESSION=0, reference is made to the current value. If EXPRESSION=1, reference is to the previous value, etc. All references are relative to the current element, which is updated each time an assignment is made to an ANALOG variable without the delay indicator attached. The maximum depth of storage of past values is declared by specifying DELAY(N) in the declaration for ANALOG variables. N states the total number of values to be saved, including the current value and N-1 past values. If the DELAY specification is omitted, N defaults to one (i.e., no previous values are saved). A DELAY specification may be included in declarations for DISCRETE variables as well, functioning as for ANALOG variables.

The MAP attribute provides a way to name contiguous fields (bit-strings or character substrings) of an INTEGER or CHARACTER variable.For each variable, up to 16 possibly overlapping fields of any positive length may be specified. This feature enables CRASH users to refer conveniently to specific bits or characters by a simple identifier instead of with a field description (see description of sub-unit selection in section 5.2.5) for every reference to the field. The MAP attribute contributes greatly to program readability when the programmer is using for example, a single INTEGER variable as a set of control flags, or is manipulating the various bits of the train control registers. A complete description of the MAP attribute is given in Appendix B: CRASH Users Manual, chapter 2.

The INTERNAL and EXTERNAL attributes are used to signify that a procedure label rather than an ordinary variable is being declared. The attribute must, of course, match the type (internal or external, section 5.2.2) of the procedure whose label is being declared. These two attributes are legal for all data-types, except that the INTERNAL attribute is illegal for

TASKs which must be EXTERNAL.

The explanation of the AUTOMATIC, STATIC, GLOBAL and INITIAL attributes is postponed to section 5.2.5: Storage Allocation.

5.2.4 Run-time Variable Checking

Since program failures are often caused by out of bounds references on arrays or delay variables, the compiler can generate run-time bound checking for array subscripts and DELAY variable past value indicators. This checking, which can be selectively enabled for any or all variables, will generate a run-time warning message or cause an interrupt with associated special condition processing (to be described later) if an out of bounds reference is detected. Since all DELAY variables are implemented as a circular list, wraparound will occur when a DELAY variable becomes full, and the Nth past value will be discarded making room for the new current value thus keeping N-1 past values and the current value available at all times. If some action is desired after the first N samples have been accumulated in a DELAY variable, a DELAYFULL condition may be specified for ANALOG or DISCRETE variables to cause an interrupt with associated special processing when the first storage wraparound is about to occur.

5.2.5 Storage Allocation

Three storage allocation methods for variables are available in CRASH: AUTOMATIC (useful for recursive procedures, or for large arrays to hold temporary results); STATIC (used when variable contents must be preserved from one invocation of a procedure to the next); and GLOBAL (useful for inter-procedure communication or "common" data areas).

The way in which storage for each variable is allocated can be controlled by the use of the AUTOMATIC, STATIC, and GLOBAL attributes in the variable declarations. If no storage allocation method is specified, AUTOMATIC is assumed, except that in the MAIN procedure, STATIC is the default and is used even if AUTOMATIC is specified.

Storage for AUTOMATIC variables is allocated dynamically whenever the procedure in which the variable was declared is activated, and freed for re-use when the procedure is de-activated. If multiple activations of a single procedure exist (this will be described later), a separate storage area for a given AUTOMATIC variable exists for each activation of that procedure. This allows recursive procedures to be written.

STATIC variable storage is allocated when the program is loaded and remains allocated when the program is exited. Even if multiple activations of the same procedure exist, only one

storage area exists for a given STATIC variable declared within that procedure. Variables should usually be declared as STATIC since there is less processing overhead involved with allocating and referencing them than there is with AUTOMATIC variables.

Storage for GLOBAL variables (which must be declared in the MAIN procedure) is allocated only in the MAIN procedure, and in the same manner as for STATIC variables. GLOBAL variables behave exactly like STATIC variables, except that they may be referenced by any other procedure, internal or external, in which they are declared. All references to GLOBAL variables are resolved by the linkage editor when one MAIN procedure and any number of external procedures are linked into a single load module. GLOBAL variables are mainly used when more than one EXTERNAL procedure must reference the same data. Consider as an example a train control program with a MAIN routine,a photocell interrupt routine, a user console communication routine, and other routines. In this case, the photocell interrupt routine would need to update a train position table and the console communication routine would need to access the same train position table to answer train position queries from the user. This situation is handled conveniently by declaring the train position table as a GLOBAL variable in the MAIN routine, in the photocell routine and in the console communication routine, thus providing a common data area for these routines. GLOBAL variables are also useful for communication between EXTERNAL procedures (e.g. if more than one return value must be passed to the caller by a subroutine, this can be done with GLOBAL variables).

An INITIAL attribute to specify desired initial value(s) may be included in a declaration to provide for initialization of storage. GLOBAL variables may be initialized only in the main procedure. If the INITIAL attribute has been used for an AUTOMATIC variable, that variable is initialized each time storage is allocated for it.

5.2.6 Arithmetic and Logical Operations

The usual PL/I arithmetic and logical operations and order of precedence are utilized in CRASH. Mixed mode expressions are allowed, with conversion between data types occurring automatically as required, except for character to numeric conversions which must be performed explicitly with builtin conversion subroutines (see section 5.2.10).

CRASH can also reference or assign values to individual bits or groups of bits of INTEGER or BIT variables. This referencing or assigning values to portions of variables is known as sub-unit selection. Subunit selection information (specified as [start,length] or [start] with remaining length implied) is appended to the variable name if desired. Substring selection on CHARACTER variables is specified in the same

manner, with characters (8 bits) rather than single bits being selected.

5.2.7 Control Constructs

The CRASH language provides a rich variety of control statements to facilitate structured programming. CRASH control constructs can be classified into two general groups, DO structures and branching structures.

The DO structures available in CRASH are:

```
DO; <body> END;
     (simple do group)
DO control_variable=initial TO final; <body> END;
     (iterated do group)

DO control_variable=initial TO final BY increment;<body>END;
     (iterated do group with increment control)

DO control_variable=value1,value2,...,valuen; <body> END;
     (stepped do)

DO WHILE <condition>; <body> END;
     (do while)

DO UNTIL <condition>; <body> END;
     (do until)

DO CASE <case
expression>; <case0>;<case1>;...;<casen>; END;
     (do case)
```

The <body> may be any CRASH statement or group of statements including another DO construct (nesting of DO groups to an essentially unlimited depth is legal).

The simple DO group is used whenever a sequence of statements is to be considered as a single statement. This is sometimes needed for IF...THEN statements (explained below) and is also helpful to enhance program clarity using a labeled (see below) simple do group.

The iterated DO group is used to execute the <body> repeatedly for different values of the control_variable. Since it is sometimes useful to perform reverse iterations, or to increment by some value other than 1 (the default increment value), the iterated do with increment control has been included. Both of these DO constructs function as in PL/I.

The stepped DO is an extension of the iterated DO. In this

type of DO group, the <body> is executed as many times as there are values in the list, with the control_variable being set to each successive value in the list for successive iterations.

The DO WHILE provides repeated execution of the <body> as long as the specified <condition> remains true. This construct also functions as in PL/I.

The DO UNTIL construct is used to execute the <body> until the specified condition is true. Since the <condition> is tested after the <body> is executed, the <body> executes at least once.

The DO CASE is one of CRASH's most powerful control constructs, allowing the selection of one of several statements for execution. The DO CASE could also be classified as a branching construct, because it provides the ability to perform an n-way branch. A 4 way branch to take care of 4 possible interrupts is:

```
INTERRUPT_SERVICE: DO CASE(INTERRUPT_CODE);

   /* case 0 - TTY interrupt */
   TTY_INTERRUPT: CALL TTY_INT_HANDLER;

   /* case 1 - paper tape reader interrupt */
   PTR_INTERRUPT: CALL PAP_TAP_READ_RTN;

   /* case 2 - fire alarm */
   FIRE_ALARM_INTERRUPT: CALL FIRE_ALARM_HANDLER;

   /* case 3 - burglar alarm */
   BURG_ALARM_INTERRUPT: CALL POLICE_DIALER_RTN;

END INTERRUPT_SERVICE;
```

If the <case expression> evaluates to a non-existent case (e.g. INTERRUPT_CODE=17 in the above example), a branch to the END of the do case is taken.

Any CRASH DO construct may be labeled like any other CRASH statement. If a DO statement is labeled, its matching END statement must be identically labeled. Labeling DO statements greatly enhances program clarity, and aids the programmer in matching DO's with END's as the CRASH compiler will check for a label match on corresponding DO's and END's. The use of labels is shown in the code example above.

The CRASH branching statements available are:

```
EXIT  DO;
EXIT  DO label;
NEXT  DO;
NEXT  DO label
```

```
GO TO label;
IF <expression> THEN <statement>;
IF <expression> THEN <statement> ELSE <statement>;
```

It is sometimes desirable to exit from a DO group before its normal completion. CRASH provides the "EXIT DO;" and "EXIT DO label;" statements to avoid the poor programming practices of using a GO TO, or altering the control variable or <condition> expression value. Execution of these statements causes a branch to the first statement following the end of the current ("EXIT DO;") or named ("EXIT DO label;") DO group.

The "NEXT DO;" and "NEXT DO label;" statements provide a convenient means to terminate the current iteration of the <body> and resume with the next iteration, providing that the condition for executing the DO group is still satisfied. This statement applies to either the current or some other named DO group as for the "EXIT DO" statement.

A GO TO statement is included to avoid certain awkward situations although its use is discouraged.

Two forms of the IF statement are possible in CRASH:

```
IF <expression> THEN <statement>;
IF <expression> THEN <statement> ELSE <statement>;
```

These statements function exactly as in PL/I. The use of IF statements to structure programs, instead of the EXIT DO, NEXT DO and GO TO statements, is encouraged as it leads to more readable code.

5.2.8 Tasking and Timing

It is sometimes desirable to have many different activities take place concurrently within the computer. Normal procedure calls cause suspension of the calling program until the called procedure has RETURNed. To facilitate real time control activities, CRASH allows the possibility of having several procedures active simultaneously, without requiring the completion of one before another can execute. These special procedures, which can be active independently of and concurrently with other procedures, are called TASKS.

A TASK can be scheduled to execute in a variety of ways. It can be synchronized with the clock, with the procedure that first invoked it, or with another procedure. It may even be scheduled to execute asynchronously when a special event or condition occurs (triggered by some internal event such as out of bounds array reference, by some external event such as a photo-cell connected to an interrupt port, or by I/O completion). These special events are described in more detail

in Section 5.2.9, Interrupts and Special Processing Conditions.

Since some jobs performed by TASKs may be more important than others (e.g., shutting down the gas supply upon detection of boiler overpressure as opposed to servicing the paper tape reader) a method for deciding which TASK should be run at a given time is provided. This is accomplished by including a priority specification each time a TASK scheduling statement is executed. The priority specifies the TASK's importance and timing requirements in the collection of programs being executed. A priority is specified by including PRIO(<N>) as a part of a CRASH scheduling statement. Where <N> is an integer from 1 to 250, or an INTEGER variable whose value is in that range. Priority 1 is the lowest, signifying the least important TASK. The MAIN procedure and all other normal procedures (i.e., non-TASK type procedures) run with a priority of 10. A TASK may pre-empt another TASK or PROCEDURE if its priority is higher than the priority of the one currently executing.

The six basic TASK scheduling statements available in CRASH are:

```
                    START   <task>      PRIO(n);
AT     <time>       START    <task>     PRIO(n);
IN     <time>       START    <task>     PRIO(n);
EVERY  <time>       START    <task>     PRIO(n);
ON <condition>      START    <task>     PRIO(n);
                    CANCEL   <task>;
```

The <time> of the form (N units) may be specified in minutes (N MIN), seconds (N SEC), in milliseconds (N MSEC) or in 100-microsecond units (just N). A <time> specification may also be an integer variable whose value is assumed to be the desired <time> in the specified units (e.g., INT_VAR MSEC).

START <task> will activate the TASK referred to by the name <task>.

AT <time> refers to time past midnight on the system clock when the <task> is to be STARTed.

IN <time> refers to time from present time when the named <task> is to be STARTed.

An EVERY statement will activate the named <task> immediately and reactivate it every time the interval specified by <time> has elapsed, until CANCELed.

A task may also be scheduled to START asynchronously upon the occurrence of some special event (<condition>). These special events are described in Section 5.2.9, Interrupts and Special Processing Conditions.

CANCEL <task> cancels the named <task> immediately if it is inactive, or upon completion if it is currently active (for self-cancellation).

A LOCK statement is provided which allows a TASK to continue execution until an UNLOCK statement is executed, or normal execution termination occurs. This prevents other tasks of higher priority from pre-empting the task issuing the LOCK statement, even though some significant time, condition, or external event may have occurred. TASKs which are unable to begin execution immediately, or which are pre-empted TASKs are automatically queued for execution or continuation later.

Since one may wish to start a TASK upon the occurrence of more than one event, more than one definition of the same TASK may occur. A method of distinguishing the occurrence of a TASK from other suspended activations of that TASK is included. Whenever a task is scheduled, an integer variable must be surrounded by parentheses and appended to the task name (e.g., TASKNAME(TASKID) where: TASKNAME is the name of a TASK, and TASKID is an integer variable). The scheduler then returns a unique identifier (in the INTEGER variable) for that particular definition of the TASK, thereby allowing the various definitions to be distinguished from one another. Any TASK that can execute concurrently with itself must be made re-entrant, and must use AUTOMATIC variables.

5.2.9 Interrupts and Special Processing Conditions

Special events that may cause a TASK to be invoked are divided into two classes.

The first class is the internal processing event which includes subscript or delay indicator range errors (SUBSCRIPTRANGE and DELAYRANGE) or the filling of a DELAY variable (DELAYFULL). These conditions were described in Section 5.2.6.

The second class of special events is the external event occurrence. There are two events in this class, IO-RETURN and INTERRUPT.

An IO-RETURN occurs when an input-output unit signals the computer that an I/O operation has completed on a specified unit (LDN) with a particular return code (RC). An ON IO-RETURN (RC,LDN) statement allows scheduling of a task whenever an I/O completion with a particular RC and LDN occurs. These return codes can indicate error, end-of-file, end-of-disk, successful completion, etc.

External INTERRUPTs can occur for a variety of reasons which vary from device to device. There are usually two possible interrupts associated with each device, A and B. When it is

desired to start a TASK because of an interrupt, the statements:

```
ON INTERRUPT_A(LDN) START <task> PRIO(N)
or
ON INTERRUPT_B(LDN) START <task> PRIO(N)
```

are used. Consult Appendix B (OSWIT User's Manual) for a description of devices and possible interrupts associated with them.

5.2.10 I/O Statements

CRASH supports three forms of I/O to enable the user to communicate conveniently with various devices. The first form is intended primarily for communicating with the console device and human operators. A second form is used to send and receive data from external devices such as D/A converters or train interface control registers. The third form is used for doing record type I/O with floppy disks, MTS, the user console, or any other device which supports record I/O.

Since CRASH was designed for real time applications, most of the I/O statements only start the I/O operation, thus allowing overlap between I/O device operation and computation. Two methods of determining when an I/O operation is complete are provided. A WAIT statement may be used to suspend the currently executing procedure or TASK until the I/O operation is done. Alternately, an ON-<condition> statement can be executed to start a TASK upon a specified return code from the operation. This allows the TASK or procedure to continue execution simultaneously with the I/O operation.

The first form of I/O consists of three pseudo-variables: INPUT, CARD, and OUTPUT. INPUT and CARD may be used anywhere any other CRASH variable may be used, except that they may not be the object of an assignment. The identifier INPUT is used to read character,integer, or real constants from the console. Whenever INPUT is referenced, one constant is read from the console input buffer, converted to the proper data-type if necessary, and transferred or used as specified. Some examples of the use of INPUT:

```
CHAR_VAR=INPUT;
IF INPUT='STOP';
INTEGER_VAR=INPUT;
```

If the console input buffer is empty, the user is prompted for a new input line. Constants may be entered several to a line, delimited by commas or by one or more blanks.

CARD is used to read one complete line from the console device. The user is prompted for an input line, which is read in as one complete character string, with no conversions performed.

The use of CARD does not affect the console input buffer.

The pseudo-variable OUTPUT is used to write a line to the system printer, usually the DEC writer console. OUTPUT behaves exactly like a simple CHARACTER variable, except that it must always be the destination of an assignment. No substring selection may be performed on OUTPUT. Conversion from numeric or bit data-types is performed automatically if necessary.

The second form of I/O is through GET and PUT statements. The GET and PUT statements are used for communication with external devices (e.g. A/D or D/A converters, Train control registers, etc.). All input or output of data with GET or PUT statements is performed by the device assigned to the LDN(s) (described in section 5.2.3) associated with the variable(s) in the <varlist>. The form for GET and PUT operations is:

```
GET    <varlist>;
PUT    <varlist>;
```

where <varlist> refers to either a single variable or a list of variables separated by commas. The SCALE, OFFSET, and CLAMP attributes (section 5.2.3) are applied during GET and PUT operations. If an LDN was not specified for a variable, the console device is used.

The third form of CRASH I/O is the GET RECORD and PUT RECORD statements. These statements have the form:

```
GET  RECORD(LDN)    <varlist>;
PUT  RECORD(LDN)    <varlist>;
```

LDN and <varlist> are the same as in the GET and PUT statements. Up to 255 bytes may be transferred for each variable in a GET RECORD or PUT RECORD statement. No conversions are performed. The data are simply transferred byte by byte as is. These statements merely initiate the I/O, allowing processing to continue in parallel with I/O operations. This is the fastest type of I/O available in CRASH, and is useful for data buffering, such as would be needed in a data acquisition program.

I/O operations are generally quite slow compared to CPU operating speed. Because of this, most CRASH I/O statements only start the I/O operation. If it is necessary to suspend processing until the operation is complete, a WAIT statement may be executed. The WAIT statement has the form:

```
WAIT  FOR  LDN,LDN,...;
```

where "LDN" is the logical device number of the device to wait for. Execution of a WAIT statement causes program execution to be suspended until the specified device signals (with an

interrupt) that the I/O operation is complete.

5.2.11 Predefined Functions and Subroutines

a number of commonly used operations are available in the form of predefined functions or subroutines. These operations include trigonometric functions, length of character strings, absolute value, random number generation, matrix manipulation and conversion, character to numeric conversion and routines to interface to the operating system for disk I/O, file manipulation, etc. For a complete description see Appendix B: CRASH Users Manual, chapter 11.

5.2.12 CRASH Summary

This section has provided a description of the CRASH language, concentrating on the features unique to CRASH, and especially on those features designed for real time applications. For more detail on the CRASH language, or for sample programs, refer to the CRASH MANUAL in Appendix B.

6. INSTRUCTIONAL APPLICATION OF FACILITY

6.1 Use of Facility

The electro-mechanical analog facility has been used for instructional purposes since January, 1976. During that time approximately 400 students have taken the course for which this facility was developed (ECE/CICE/IOE 469, Real Time Computing Systems). In this section we will describe the current version of the course including the laboratory projects, evaluations of the facility by students and staff, and indicate suggestions for improvement. The course consists of three hours of lecture and one three hour laboratory period each week.

6.1.1 Course Objectives and Material

The major objectives of this course are:

1. To provide experience in programming a real-time microcomputer system that includes synchronous and asynchronous interrupts.

2. To provide experience in the application of A/D and D/A converters to real time problems.

3. To provide experience with practical data sampling and frequency analysis with Fast Fourier Transforms. 30

4. To introduce basic concepts of digital process control and its application.

5. To provide experience with distributed sensor systems in a dynamic input-output environment.

A detailed course outline is provided in Appendix F. The laboratory portion of the course is well-integrated with the lectures. In addition, material specific to each laboratory exercise is provided at the beginning of most laboratory periods. The four projects in the laboratory are summarized below; the experiment descriptions are given in Appendix G.

6.1.2 Standard Projects

6.1.2.1 Project 1. String Reverser

Each student is asked to write a CRASH program to accept a character string from the Decwriter keyboard and type out that string in reversed order. This short project is designed to allow each student to become familiar with the CRASH language and some features of the operating system (OSWIT). One week is devoted to this project.

6.1.2.2 Project 2. Data Acquisition

Each student pair develop a CRASH program to sample an analog signal and store the sampled data in a specified form on a floppy disk file. They adapt a command handler to accept specific command words from the keyboard (with error checking) to specify and control the sampling process. The input signal is a low frequency "respiration" signal that results from breathing through a hollow tube that contains a thermister. After the data are acquired and stored in the floppy disk file, they are copied to a file on the central computer where a fast Fourier transform (FFT) is performed. The transformed data can then be plotted on a graphics terminal (Tektronix 4010) and/or listed on the system line printer for interpretation. This project provides experience with command handlers, data sampling and synchronous interrupts, file writing, and frequency domain analysis via the FFT. Three weeks are devoted to this project.

6.1.2.3 Project 3. Servo Controller

Each student pair develops a CRASH program to control the response of a servo motor to a step input command from the keyboard. Two control algorithms are implemented, Proportional Integral Derivative (PID) control, and velocity feedback control. The amount of proportional, integral, derivative or velocity control is specified from the keyboard as is the sample period and the number of periods of feedback delay. One hundred samples of the position response to a step input are stored in a buffer. By listing or plotting the response data on the 4010 graphics terminal, the students can observe and study the servo motor response to varying types and amounts of control, including those combinations that result in unstable responses. In the process they have to relate z-transform representations of sampled data systems to the difference equations for the controller and the plant (the servo system and its power amplifier). Four weeks are devoted to this project.

6.1.2.4 Project 4. Electric Train Control

Each student pair develops a data structure to describe the layout of an N-gauge model train setup. About seventy sections of track must be specified in such a way that power (voltage) of correct polarity is applied to each track section in anticipation of the movement of the engine onto that section. The layout, shown in Appendix D, also includes 64 switches, one crossover, one reversing loop, and 64 photocell pairs to sense the presence or absence of the engine and associated cars. The basic problem is to control a single engine train in response to speed, direction and switch commands given from an analog control box. The students have to deal with multiple asynchronous interrupts in an electrically noisy environment. Project options include programming for speed control on various track sections (i.e., slowing around curves) multiple train

control, and switchyard interactions. Five weeks are allowed for this project, which is perhaps the most complex and popular exercise.

Each group demonstrates the program developed for each project to the instructor who then checks for various features and failures. A written report is submitted by each group a week later in which the data acquired are analyzed and discussed. Grading of the work is based both upon operability and system design.

6.1.3 Independent Study Projects

One of the most beneficial aspects in the development of the real time computer applications laboratory has been the heavy involvement of undergraduate and graduate students in independent study projects. In fact, most of the hardware and software facilities in the laboratory were developed in this manner. The results of these projects have been extremely valuable to the development of the laboratory and have provided an outstanding educational experience for the students involved. The students gained design and development experience they would not normally have acquired until their first job. As a result they not only have greater experience to carry with them to their employers but have developed some maturity in dealing with real projects.

Many of the projects were quite substantial and involved more than a single student and several terms worth of work, sometimes beginning with one set of students and ending with another. To illustrate the sort of tasks undertaken the major projects are listed below:

1. Design and implement a basic real time operating system for the LSI-11 (the basis for OSWIT).

2. Design and construct a floppy disk controller for the Memorex 651 floppy disk drive used in the laboratory.

3. Design and implement a disk file system for the floppy disk.

4. Design and implement communications software to support the MCP protocol for communication between the LSI-11's in the laboratory and the university's central Amdahl computer.

5. Design a higher level language to support real time control applications.

6. Implement a cross compiler for the language designed in number 5.

7. Design and implement a high level debug package (RAID) to be used with CRASH.

8. Design and implement a resident assembler (assemblies are normally accomplished on the Amdahl computer with the object code then downloaded to the LSI-11).

9. Design and implement a resident linkage editor for the LSI-11.

10. Design and implement the hardware controller for the N gauge model railroad system.

11. Implement a distributive processing system between one of the LSI-11s and the University's Amdahl computer.

12. Study advanced control concepts with higher order systems simulated on the analog computers.

Many of these projects were quite substantial and extended over long periods of time. The CRASH language and compiler, for example, were designed in a single term. Implementation, however, extended over a two year period to achieve a reasonably well debugged compiler.

These projects have been of great value to the participating students. First of all the students had an opportunity to become involved in real projects. This gave them exposure to practical problems which they can expect to encounter in industry. They had to work within a group of people. There were time constraints involved. Finally, they had the satisfaction of seeing their product actually being used by other people. There were also substantial academic benefits as well. First of all the experience gave them a much fuller understanding of the basic technique they had studied during their regular course work. Moreover, much of the software developed was state-of-the-art. They received exposure to advanced techniques and new ideas.

The student reaction to these projects has been excellent. The students involved almost to a person exhibited extreme enthusiasm for their projects and worked with a fervor I have seldom seen in industry. In addition to producing useful products they achieved considerable personal satisfaction. Individual comments received from them at the completion of their projects indicate that they felt their activities were well worth the time spent on them.

6.2 Reaction to Use of Facility

The course based upon the real time computer applications laboratory, CICE/ECE/IOE 469, was begun in January 1976. It has been offered three times a year since then. The CRASH language was introduced into the course during the fall of 1977.

Each term the department evaluates its teaching assistants and instructional laboratories. Evaluation data on the course is available for the fall and winter terms since 1977. Four types of evaluation statistics have been collected: on the laboratory, on the course comparison to other engineering courses, on the course in comparison to other university courses, and on the CRASH language. On the average the 469 laboratory ranked third among laboratories.

Prior to 1978 the College of Engineering conducted its own course evaluations on its courses. These evaluations were on a scale of 0 to 4. The table below shows the three terms of statistics available on 469 in comparison with the average scores in the Electrical and Computer Engineering Department and the Computer, Information and Control Engineering Program.

Term	469	ECE Average	CICE Average
Winter 1977	3.20	2.42	2.67
Fall 1977	2.52	2.28	2.63
Winter 1978	3.17	2.48	2.49

Beginning with the Fall Term 1978 the course evaluation was shifted to match a University-wide evaluation which is based on a 1 to 5 point scale. During the 78-79 academic year the evaluations were as follows:

Term	469	University Median	25% of Courses Above this Level
Fall 1978	4.45	3.96	4.28
Winter 1979	4.33	3.85	4.18

It seems evident from these ratings that the course based upon this laboratory material ranked well in the upper 25% of the courses within the University (probably at about the 15% level) with the exception of one term. The exception occurred during the fall 1977 term which coincided with the introduction of the CRASH language to the course. During this term numerous bugs were found in the compiler and the class suffered accordingly. In addition an attempt was made to cover both assembly language and CRASH in the course which was too heavy a load. Note that in the subsequent terms as CRASH continued to be used the course evaluations rose.

During the Winter Term 1978 a student evaluation of the CRASH language was conducted. Among the questions asked was a comparison of CRASH and Fortran and CRASH and assembly language. On a scale of 0 to 4, with 4 being the preference for CRASH and 0 being the preference for Fortran or assembly language. The CRASH language received the following evaluation:

3.2 with respect to Fortran

3.3 with respect to Assembly Language

It is clear that once most of the bugs had been removed the CRASH language was highly preferred by those who had used it.

6.2.1 Instructor's View (SLB)

I gave half the lectures and taught a laboratory section of CICE/ECE/IOE 469 for the first time in the Spring Half-Term 1979. I am giving all the lectures and directing the laboratory during the fall term.

The course evaluation which follows covers both the lecture and laboratory portions of the course since they are integrally related. However, the laboratory portion of the course that relates to the train control project will be discussed in greater detail than will other portions of the course.

The course evolved to its present form through the dedicated efforts of Professor R. A. Volz. Several undergraduate and graduate students have made significant contributions in both hardware and software areas over a period of several years. Therefore I was able to step into the course after many of the start-up problems had been solved and the laboratory projects were well-defined. However, I had to learn the software (CRASH), the LSI-11 specifics, and the real time aspects of the course essentially as a student, and without benefit of the historical perspective gained by Professor Volz.

I believe the course to be one of the best offerings available to students with computer and control interests. The course combines elements of software (languages and programming) with hardware (sampling, conversion, logic design) and control (sampled data systems, stability, types of control) from a users point of view. Thus each student must combine and use computer and engineering skills in creative and practical ways. In general I believe that the course is excellent.

There is a high level of student interest in the course as indicated by the fact that the course fills up so fast that many (30-40) students end up on the "wait" list each term. This is particularly impressive since the course has a reputation for being very time consuming and although 4 credit hours are given it is considered as the equivalent of a 5 or 6 credit course.

Unfortunately because of the high demands placed on the students the attrition throughout the term may be as high as 20%. This term we have lost about 10-12 students from an initial group of 65.

The problems associated with the course fall into two major categories, laboratory facilities and curricular. The facilties problems will be discussed first.

6.2.1.1 Facilities Problems

The laboratory faciilties are limited to 3 LSI-11 based setups. Each student is enrolled in a laboratory section that meets formally for a 3 hour period each week. There are 13 students in each laboratory section at the beginning of the term and 10-11 by the end of the term. The students work in groups of two or three (occasionally). Each student can gain access to the laboratory at other times through about 10 hours of laboratory instructors office hours outside of regularly scheduled laboratory periods. Furthermore, if they pass a "key" test they can obtain a key to the laboratory for evening and weekend access. However, even with all this apparent access to the laboratory facilities, the class enrollment is so big that the students have difficulty in "getting on the machines", particularly near the project demonstration deadlines. Furthermore, the intensive use leads to increased equipment malfunction and student frustration. The solutions are obvious: reduce the number of students or increase the number of set-ups. Unfortunately space and financial considerations have made it difficult to increase the facilities and the pent-up demand has made it impossible to reduce the enrollment.

Each laboratory section is reasonably well equipped to handle 9-10 students working in pairs which seems to indicate a total course enrollment of about 40. The current large laboratory sections make it difficult for the instructor to service all the programming and operational problems that occur, particularly at the beginning of the term.

The train set-up has problems. An N-gauge system is clearly an appropriate choice in terms of the size required for a reasonably challenging train control exercise. However, the actual components for N-gauge systems leave a lot to be desired. The track sections do not match well which leads to derailment problems. The track must be cleaned often to reduce the contact resistance to the engines. The switches are not well made and thus have to be adjusted often to make them work. The engines are not particularly reliable in an intensive use situation. The net result is that the train set-up must be prepared and tested before each demonstration and at various intervals during the duration of the train experiment. Unfortunately there is no obvious solution to these problems except to move to a larger guage system which would be very costly in construction time and

would require a significantly larger area for the layout.

The connection of each set-up to the central computing facility (MTS) via the remote data concentrator works very well. In general MTS is quite reliable. On the other hand progress on the laboratory exercises can be severely compromised on those occasions when MTS is not operating. I would like to sever the umbilical cord to MTS but this would necessitate a small dedicated "central" computer system in the laboratory to service the editing, compiling, and listing functions of MTS. Since over $30,000 of MTS computer funds are expended for this course each year there may be sound economic grounds for an in-house "central" computer.

6.2.1.2 Curricular Problems

Perhaps the most significant problems with the course are in the curricular area. The normal prerequisite for the course is a course in Fortran and junior standing. However, the course is also taken by advanced graduate students. Therefore, at any given time the class consists of students who are both naive and sophisticated in their software, hardware, and systems background. The spread of backgrounds is so broad as to compromise the teaching and learning situation. The naive student is introduced to new material on a continuing basis whereas the sophisticated student may already have had major portions of the lecture material in other courses. There will continue to be problems of this type as long as a single course if offered. However, the prerequisite should be changed to indicate the necessity for more background and maturity than is presently required. I have attempted to handle this problem by warning the naive students on the first day of class concerning the course content and "real" prerequisites. Essentially each student should have taken at least one out of 5 or 6 other computer, mathematics or systems courses that provide background or lecture level material for this course. Then the amount of new material would be manageable for the less well prepared students.

The wide differences in background make the presentation and selection of the lecture material difficult. It is almost impossible to avoid "snowing" one group of students and boring another group with the same material in many cases. To some extent almost everyone is "snowed" or "bored" at some time during the course depending on their backgrounds. Perhaps this will always be the case in any course that tends to put together a wide variety of techniques to solve a relatively broad base of problems.

A reasonable solution would be to offer a two course sequence, each with a laboratory. The first course would be aimed at the "naive" student with perhaps only the present Fortran prerequiste. The lecture material would concentrate on

the CRASH language and simple concepts of data sampling and conversion. The second course would cover details of real time programming and control as well as more advanced material on data sampling, conversion, sampled data systems and asynchronous control. Computer to computer communication and interfacing would also be covered from theoretical and practical points of view. The first course or background in structured programming and PL/1 or PL/C and at least senior standing would be an appropriate background for the second course.

Again, the major limitations in effecting any changes here are lack of availability of staff and funds.

6.2.1.3 Textbook Problem

There is no adequate textbook for this course. The practical approach taken to the implementation of real time computer control has forced the presentation of a set of topics that are not covered in any single text. The CRASH language and OSWIT operating system are described in manuals not specifically designed as teaching material. They contain few useful examples and are more in the nature of a catalog of what is available. Also they are specific to LSI-11 based systems in communication with MTS. The other lecture material is generally basic material from several major areas that are not treated in any single text. Finally, the specifics of the LSI-11 and how it is connected in our laboratory can be found only in the Digital Equipment Microcomputer Handbook or in assorted circuit schematics. Presumably much of the material developed for this report will become availble for student use in the near term. However, it would have to be revised somewhat before being given to the students.

The volume of unpublished material distributed in this course is immense. In addition to the CRASH and OSWIT manuals the lectures are distributed in advance so that the students do not have to copy large amounts of information from the blackboard. Hopefully, this allows them to concentrate on understanding the material presented rather than on transcribing it. Unfortunately, some students depend on the written lectures rather than attend class. However, students who already know the material presented in a particular area may actually be able to use the lecture time more profitably for other endeavors.

6.2.1.4 Conclusion

The evaluation and course details provided here may help to place the electro-mechanical analog facility in its educational context. I have enjoyed the learning and teaching experience associated with the 469 course. It has helped me to appreciate the many problems associated with computer based real time control and some of the methods for solution.

6.2.2 Students' Views

Individual student reactions to the laboratory and course have varied widely according to the background of students taking the course and the administrative conditions under which the course has been offered. It has been necessary to allow 60 people in the class at one time even though there are only three laboratory setups. Even with this course loading, it has been necessary to turn away a substantial number of people each term. In the following sections, views from individual students with both favorable and unfavorable reactions to the class are presented.

6.2.2.1 View 1 - - Jack Wenstrand

CICE 469 is a course in the application of real time computer systems. It is described in the College of Engineering Bulletin as :

> Principles of application of real time computer systems to engineering problems. Topics include: computer characteristics needed for real time use, mini/micro computer operating systems, man-computer communication, basic digital logic design, analog signal processing and conversion, and inter-computer communication. Topics investigated via laboratory using microprocessor system.

My background coming into this course included one semester of structured programming, and a semester of circuit analysis. This should be a minimum requirement for the course. I was able to understand the material presented in the course, but everything was new to me. I found the course to be interesting, informative, worthwhile, and very difficult.

The lecture opened with an introduction to real time computing. Here the traumatic notion that a computer program need not execute in the same sequential order in which it was typed first entered my sophomoric mind. Approximately the next two weeks were spent easing me past this critical moment as the CRASH compilers were thoroughly discussed in class and the concepts behind the real time functions that it implemented. This discussion was important, as many of the students in the class had no background in real time operations.

At this point, the language itself merits a few words. The CRASH language was a valuable tool in the laboratory. A high level language much like PL/1, it enabled many students with limited programming experience to construct fairly complex real time application programs. If the same programs were to be implemented on the assembly language level, a much stronger programming background would be required to make it through the course. The main drawback to the language was that the compiler still had some bugs. If you hit one of those bugs, and CRASH

produced code that would not execute, you had little recourse but to restructure the code and try again. The course would not have been possible for me without CRASH. It was sometimes rather frustrating, but definitely necessary.

The next topic covered was the LSI-11 computer itself: everything from bus structure to memory mapped I/O. Special attention was devoted to the interrupt structure. This was important because the major part of real time computing is processing interrupts. The discussion of I/O was also very appropriate, helping us to understand how real information in the real world can be converted to and from the binary bit patterns that the computer crunches.

The next topic considered was that of digital to analog and analog to digital conversion. This was approached, as it must be, from a circuit analysis point of view. After a brief introduction to op-amps, we were quickly pushed through D/A converters and on through successive approximation A/D converters. Some of the more software oriented people in the class found all this rather difficult, but to me it was fascinating.

Then came sampling theory. These lectures were the basis for the first major lab experiment. The Fourier Transform methods and results of its numerical implementation were covered, including the effects of sample rate on results, aliasing, and use of filters to improve results.

There was no break in this course -- the material just kept coming. Next, the professor dove right in to control theory. Wait a minute! What's a Z-transform? This section of the course was probably the most difficult for me to understand. The whole concept was completely new to me, and I think the same was true for the rest of the class. Perhaps a little more time could have been spent here. While this was going on, we were working on a servo controller in the lab. This was helpful from the standpoint of allowing us to test some of the concepts that we learning in lecture.

Computer-computer communications was another very interesting topic that was covered in this course. I am glad that that was in the course, as it is becoming more and more important as the trend towards digital signal processing grows. Topics discussed included requirements for a data communications protocol, a couple of protocols now in use, and noise and error detection/correction considerations.

The lectures concluded with a survey of microcomputers now on the market. Costs, features, and capabilities were compared and contrasted. As cost is such an important factor in any real world application, the course would not have been complete without this section. The lecture encompassed all of the topics

necessary as a basis for real time systems design. Each of the different subject areas was covered in sufficient depth to be of practical value. All of the topics were important to the course, and I hope that none of them will be omitted in the future. The lab and the lecture were very well coordinated, each re-enforcing the other.

The first lab project was a simple problem which served to introduce the class to CRASH. We were to program the LSI-11 to input a string from the terminal, reverse it, and output it to the terminal.

Our next assignment was to write a general command handler. The command handler was necessary as the following experiments were to be command driven. This project had its good points and its bad points. Writing the command handler required a lot of time, especially for those with minimal programming experience, and has no direct relation to real time. On the other side of the coin, the experience forced people to familiarize themselves with CRASH before entangling themselves with new real time concepts and constructs. I feel that the project was worthwhile, but could probably better be replaced by another real time experiment.

The next experiment involved synchronous sampling of data and analysis of that data. The hardware included essentially a thermister wired through an op-amp to a A/D converter. This provided us with an eight bit number proportional to the thermister temperature. We generated the waveform for sampling by breathing on the thermister. We were to record a specific number of bytes on our floppy disk in a very specific format. Also required was a routine to unpack the data bytes and print them out. After taking a data set, it could then be transferred to MTS where a fast Fourier transform program was available for our use. This allowed us to verify rather easily the sampling theory basics that we were learning in lecture. The fast Fourier transform was definitely a nice tool to use as it simplified the data analysis. However, in order to use the program we had to pack the bytes in to the disk file in a very specific manner. This part of the experiment is probably what caused the greatest amount of trouble, particularly to those without much programming experience, and again this was not directly related to real time computing. This experiment could be improved by modifying the fast Fourier transform program to accept one data point per 16 bit word, instead of one data point per byte as it does now. While this does waste a little space, it greatly simplifies the packing and unpacking of data points.

The third project was to implement two algorithms to control a servo. One was a velocity feedback controller, and the other was a proportional integral derivative controller. We were given equations for both controllers in forms suitable for implementation. That was good, in that most people in the class

did not have a sufficient comprehension of control theory to come up with the equations themselves. This experiment also re-enforced the lecture material by verifying such facts as a large integral term will make the PID controller unstable.

The last experiment was the one that everyone was waiting for. The idea of using an electric train for a lab project on handling asynchronous interrupts must have been a stroke of genius! The train attracts interest to the course, helps to keep the students interested, and is a perfect example of a case where priority handling of asynchronous interrupts is necessary. The experiment was by far the most difficult of the term, primarily because the class was on its own. We were given specifics on the train board I/O and a couple suggestions on what our data structure (which had to represent the train board to the computer) should contain, and that was it. It was a good project -- it really made me think.

The lab was an essential part of the course. While the lecture was covering sampling and control theory, which seemed more than a little abstract to me, the lab served to force me to relate these concepts to the actual processing that I had programmed the LSI-11 to do. The success of the lab was rather dependent upon the teaching assistant. As each of the problems was an introduction to a different area, the class required a number of specific suggestions of methods of approach to complete the projects. The dependence on the teaching assistant could be reduced by including more details in the write-up that is handed out for each experiment, and by devoting a lecture period to each lab or scheduling some extra class periods for that purpose.

The lab facilities were good. CRASH, as mentioned before, had some bugs, but it was definitely better than the assembly language alternative. The hardware was reliable. The train board was subject to noise, leading to some false interrupts. However, such is the case for real world systems also, so the experience of trying to program around the false interrupts was a good one. The one thing that could have been improved was availability. There were three LSI-11s and only one train board. With the amount of time and work that has gone in to that train board, I understand why they don't have two, though. Access to the train was a problem at the end of the term. Members of the class were given keys to the lab after passing a "key test." That allowed students 24 hour a day access to the lab, which made all but the last week very acceptable.

This course does not replace a control theory class, a sampling theory class, a circuits class, and all of the math necessarily prerequisite to those classes. However, it does allow a person, after one long, hard term, to be well versed as to the capabilities of real time systems, and techniques for programming them. Looking back on the course, I am still amazed

by the quantity of material covered. I have never had a class, before or since, in which I learned so much, so fast. The course was excellent. All college classes should be like CICE 469.

6.2.2.2 View 2 - - Richard Jungclas

The real time computing course has been successful in meeting the goals specified in previous sections and the expectation of students. One of the most significant aspects of the course is the real world interaction of computers and physical devices. In most college computer courses, the computer programs developed by the class consist of some contrived results from simulator and test data. In this course, the computer program has a casual relationship with sensing and controlling a physical device such as the servo or the trains. Although computer interaction and manifestation might be seen in printed results from simulations, having these interactions and manifestations occur physically imprints these interactions in student's mind.

6.2.2.3 View 3 - - Terry Rosenbaum

My reaction to the use of the train lab facilities was generally unfavorable. My reactions can be broken down into three categories: my reaction to the hardware facilities, my reaction to the software facilities, and my reaction to the real time programming class.

The hardware facilities are over used. This leads to two problems. The first problem is breakdowns. The LSI-11s, Decwriters, and floppy disks generally function quite well in spite of their heavy use. The "breath tube" peripheral also seemed to function well. The servo mechanisms seemed to break down occasionally, thus increasing the load on the remaining units. The problems I experienced with the train were non-functioning photocell sensors, worn out rolling stock and various problems with the tracks. The non-functioning photocells introduced an unnecessary obstacle into train control programming. The worn out engines, cars and tracks made program testing difficult. Engines would sometimes stall or not start when power was applied. Cars would come unattached from the engines due to worn or broken couplers. Poorly aligned or worn out tracks caused train derailments. All these things detracted from the excitement of controlling the train by computer, thereby lessening student enjoyment and satisfaction with the experiment, and giving rise to a certain amount of frustration and subsequent drop in motivation. This problem could be corrected with increased maintenance.

The other problem with overuse of the facilities is the amount of time available to students for program testing. The high usage necessitates round the clock sessions at the end of the term. This overtires the students thereby lessening their

efficiency and reducing their chances of success. Also, the amount of debugging time available is just not enough, as is evidenced by the low rate of success on the train program. Most students have their programs partially working, but few are really satisfied with their final result. The solution to this problem is obvious, and extremely expensive--purchase more equipment. I have another suggestion to alleviate this problem through restructuring of the course. This suggestion will be covered in the section on suggested improvements.

There were two main problems with the software reliability and documentation. All software seemed reasonably reliable with the exception of the CRASH compiler. A number of bugs existed in the compiler, and some of these compiler bugs caused erratic program operation not attributable to source code errors. Subsequent work on CRASH by me has removed many of these bugs, however, and this problem should be eliminated in the future.

The problems with documentation were the lack of a good operating manual for OSWIT, and some rather poor explanations in the CRASH manual. An OSWIT manual has been written which should alleviate the first problem. The CRASH manual which exists needs work. Perhaps this would be a good project for a technical writing class in the Engineering Humanities Department. Since all engineering students are required to take technical writing, it seems feasible to set up a section of 469 which would operate jointly as a technical writing class to attempt to rewrite the CRASH user's manual.

The problems I experienced with 469 were due to my programming background, and to the overly large number of students in the class. Since I had already taken many computer courses including operating systems design and compiler design, some of the program assignments were a bit underchallenging. The large number of persons in the class led to overcrowding in the lab, and caused the overuse problem explained earlier.

I feel that my most rewarding experiences with the train lab came not through the 469 class, but through my summer independent study project debugging the CRASH compiler, and my subsequent involvement in this documentation project.

Debugging CRASH provided me with experience in large scale program debugging, and greatly increased my understanding of compiler operation in general. My involvement in the writing of this document has helped to develop my technical writing skills, skills which are very important for engineers to possess. While neither of these benefits are directly related to real time programming, my work on CRASH did help to increase my understanding of real time operations.

I will now suggest some improvements which could be made to the train lab facility and to the 469 class. My suggestions for

improvement can also be broken down into three categories: possible improvements to the hardware facilities, possible improvements to the software facilities, and possible improvements to the 469 class.

For the hardware facilities, I would recommend better maintenance procedures. A maintenance schedule should be developed if one does not exist. Programs could be written to verify the operation of the train board sensors, switches, servo systems, etc. These programs should be run at regular intervals (perhaps daily during period of heavy use) and any bugs discovered should be corrected promptly. This would greatly increase the likelihood that all equipment is properly functioning and available for student use. I would also recommend the purchase of additional LSI-11s if funds, space, etc. could be found. This would help to alleviate the overcrowding in the lab.

While the CRASH compiler has been improved considerably already, there is still room for more improvement. The main problem in CRASH is stack depth checking. The LSI-11 stack is heavily utilized by CRASH programs, and stack overflows often cause erratic program operation or abnormal termination. The compiler should be modified to keep better track of stack depth during compilation, thus enabling detection of stack overflows during run time checks. Currently, the stack is often used (especially during calls to system subroutines) without upping the maximum stack depth counter (done by calling the "PUSH" subroutine). This leads to a situation whereby it is possible to overwrite the stack into the buffer region without a stack overflow being detected (because the maximum stack depth being tested against was incorrect (too low)). This causes unexplained program failure.

The BIT variable parameter size feature is not properly implemented. Currently, all BIT variable parameters must be treated as integers, because the proper field width is not passed as a part of the procedure call (this should be changed).

A possible extension to CRASH would be the implementation of TRAIN control primitive statements. This would tend to standardize train control I/O somewhat, thus aiding in debugging train control programs. The operating system, OSWIT, could be improved by the addition of better error diagnostics. Currently, error messages are very short and undescriptive. About all that a user knows after receiving an OSWIT error message is that the program bombed. More descriptive error messages would certainly aid in debugging programs. Also, a real time assembly language debugger is needed. The current assembly language debugger runs with interrupts off, limiting its usefulness. Perhaps assembly language debugging features could be added to RAID.

To alleviate the problems with the 469 class which I

described earlier, I would recommend restructuring the class. Due to the fact that the field of real time computing has expanded greatly in the past few years, I think that a one term long 469 class inadequately covers the subject of real time computing.

I think that the class should be split into two one term classes. The first term would introduce the student to the software used in the real time laboratory, and introduce the student to real time operations, concentrating on data acquisition, D/A and A/D converters and on elementary control problems such as the servo control experiment. This would give the students a more thorough understanding of D/A and A/D converters, and allow more time for learning about the software facilities in the lab before tackling more complex problems.

The second course could be a more advanced course covering interrupt programming, other types of control interfaces (parallel I/O, modems, etc.) real time operating systems (perhaps a short survey of what is available in the industry today), data communications (a very important part of distributed sensor systems), and ultimately the train control problem.

This restructuring would of course require the purchase of additional equipment (two to four more LSI-11 setups), personnel expenditures for development and instruction, and probably additional lab space. I also have some ideas for new experiments which would possibly be designed: a sound processing experiment; a sound or light tracking device (an extension of the servo control experiment to make it do something useful); an acid titration control experiment; a transistor curve tracing experiment; a logic device tester/identifier; capacitor and resistor value measuring experiment.

Some of these experiments could be implemented with minimal cost, while other would probably be very costly and time consuming to develop. The division of real time computer instruction into two courses combined with the implementation of new experiments (both to fill in the lab schedule and to be offered as alternatives with the student choosing the more appealing of say two or three experiments) would definitely make the course more interesting, more challenging, and able to offer students a more complete background in real time computing.

As for the problem of overcrowding in the labs, the division of the course may help to alleviate this problem too, although this is not certain. A less ambitious experiment schedule in the introductory course would allow more time for each experiment, hopefully alleviating the problem of not enough "hands-on" time for each student. Although it is a difficult decision to make, perhaps it would be necessary to set somewhat more realistic enrollment levels than exist at the present time.

At any rate, the problem of access to the train setup would be greatly reduced, both by the fact that there would be more time available for this experiment, and by the fact that due to natural selection process, there would probably be less students (at least less half interested and under motivated ones) in the advanced course.With more time available to students in the advanced for working on the train setup, the success rate should rise considerably, as would student satisfaction with this experiment.

I must temper the negativism of my views by adding that my position clearly represents the minority viewpoint. The CICE-469 class has fullfilled its objectives for providing students with a sound background in real time operations (and even surpassed them). If given a choice between having the course as it exists now, or not having it at all, I would have to choose the former. Also, I must add that I have nothing but the highest respect for the ingenuity and years of labor which have gone into the development and implementation of the lab facility and the course.

7. SPECULATION ON OTHER APPLICATIONS

7.1 Software Validation

Among the suggestions for use of the computer controlled train facility is the validation of certain software systems. Chuvala and Beck [1] have expressed a strong interest in facilities for validating large complex software systems. The discussion in this section is speculation as to what might be done. It in no way represents the results of extensive research, nor do the authors necessarily advocate the ideas discussed. Rather the discussion is somewhat in the nature of a brainstorming session to try to determine if there are areas in which the basic structure provided by the train system might be of use.

7.1.1 Software Engineering

The development of reliable computer software in a timely fashion at reasonable cost is one of the most significant unresolved problems in the computer area today. Software projects are almost always late, over cost budget and seldom match the original specifications. Moreover, software products are generally laden with errors that appear only after use has begun, resulting in a long drawn out and expensive maintenance operations.

Software engineering refers to a collection of techniques developed and still being developed to help alleviate these problems. The software engineering techniques addressed the following aspects of software development

1. validity--does the program function properly?
2. performance--measured in terms of execution time and storage size required.
3. software architecture--the organization and structure of the software system.
4. ease of use--the human engineering aspects of the software.
5. maintainability--how readily can the program be changed without introducing new errors in other parts of the program
6. cost--how to estimate development costs beforehand and control them during development.

The life cycle of a software product must be considered in addressing these issues. Most of these issues must be considered

across most if not all phases of this cycle. Only by considering this entire life cycle can one hope to adequately manage the software development problem.

The software life cycle can be divided into the following phases:

1. functional requirement specification and analysis
2. development of software specifications
3. software design
4. software implementation
5. validation
6. operation and maintenance.

The particular concern here is software validation even though this item is a separate phase in the life cycle of software. As pointed out by Ho [2] it is an issue which must be considered throughout the design and implementation phases as well. Ho points out that approximately 2/3 of the software errors are design errors. Furthermore, design errors are much more difficult to find and correct than are implementation errors.

There are typically three basic approaches to program validation: testing, program proving, and automated aids. Of these, the first and the third are by far the most widely used. Program proving techniques typically involve the development of a set of input and output assertions such that if the input satisfies the input assertion and the program terminates the output satisfies the output assertion. Automated theorem proving techniques are then used to verify the relation between the input and output assertions. In spite of much work on these techniques, such as the work by Stavely [3] which uses modular code structure, and higher level assertion languages to reduce the difficulty of these techniques, validation of software by program proving techniques is still a very complex problem. It is generally not feasible to use them with large complex pieces of software.

Testing forms a basis for almost all real software validation work. In fact, after a piece of complex software is put into operation the technique continues, with the users as the testers of the system. The errors they find become a large part of the maintenance of the software product. Since exhaustive testing is not typically feasible one of the major issues in testing software is the determination of a suitable set of test inputs. A number of techniques for selection of inputs have been made (see Ho [3] for a good summary of these)

including the development of formal criteria (such as forcing all program branches to be taken, Huang [4]), or that all statements be executed. However, the most common method of selecting tests is functional testing in which one identifies sufficient input to test the major functional activities of the product. For example, Goodenough and Gerhard [5] propose a methodology for selection of a test based upon decision table techniques.

Every programmer uses a variety of automotive static and dynamic testing aids whether he realizes it or not. All compilers and assemblers, for example, have some level of error detection built into them for at least proper syntax. Other checks commonly found are for matching types, undefined variables, etc. Dynamic analysis usually includes run time error checking with programs. This may include numeric, arithmetic anomalies (such as divide by zero) or may extend to address checks on data references, subscript range checking on array variables, etc. Unassisted automatic aids, however, are not yet capable of completely validating complex software. This condition will remain for the foreseeable future, since automated aids do not usually contain formalized descriptions of systems requirements and therefore cannot possibly verify the behavior of a program against its requirements.

Another area which may offer some potential for assistance is software simulation. By use of simulation it may be possible to model the system throughout its development and verify at each stage that the specification performance requirements are met. Relatively little work, however, has been done in this area (for example see Rowe [6] or Berger [7]).

7.1.2 Possible Areas of Train Utility

With this introduction to the validation problems of software engineering let us consider possible areas in which the train system might be of assistance. It is clear that there are some areas where the train is highly unlikely to be of any utility, such as in the validation of numeric computations. On the other hand there are areas where the visualization provided by the train system may be of assistance in software validation.

Perhaps the most obvious point to consider is program control flow. It is well known that control flow programs can be represented as graphs with nodes representing program statements and arcs representing flow between statements. Similarly the train layout can be represented as a graph. If the underlying graphs were identical, then the train layout would have the potential to represent control flow in a program.

It is also possible to consider modeling the control flow for tasking. Essentially one could have one engine for each task. Engines would stop and start as they are removed and

inserted to execution.

Though in general it does not seem possible to represent all operations by the train system there are some situations in which it is at least conceivable that something could be accomplished. First of all, I/O operations could be represented by the pickup and/or release of cars from sidings. Similarly different types of cars or coded cars might represent data types. The use of a train facility to represent queuing operations would be perhaps somewhat more realistic. Many large systems programming activities involve a number of queues with different strategies for managing them. Queues represented by cars on the railroad could provide an effective visualization of queuing operations.

7.1.3 Program to Train Coupling

The previous section describes some potential areas for use of the train in software validation. These were presented, however, only at the logical level. In order to actually benefit from such a system there must be a mapping from the program to the train. The program does in fact include mechanisms for operating the train. Even for an idea which seemed logically attractive if there were no mechanism for implementing the train control the idea would be of little value. This section identifies briefly some potential methods for control of the train which will be discussed further.

Most simply one could write a simulation of the program as a separate entity in which the train represented those aspects under study. At the other extreme one could envision modifying the compilers or interpreters used to automatically generate calls to drive the train. A third possibility would be to have the user insert the desired calls at run time through a suitable simulation package which has been appropriately augmented with train control functions and a macro facility.

These ideas will be explored a bit further in a subsequent section.

7.1.4 Potential Logical Relations Between Programs and a Train System

This section is concerned with potential logical relations between programming considerations and a computer controlled train system. It is not concerned at this point with physical implementation of that connection. This will be considered in the next section. Rather it is concerned with exploring the potential usefulness of such studies should physical implementation prove feasible.

7.1.4.1 Control Flow

As noted above the most obvious correspondence between a program and the computer controlled model train is that both can be represented as a graph. Truly then if the underlying graphs are identical there can be a correspondence between the program and the train. This suggests that one might try to model program control flow with the procession through the execution path of the program being represented by movement of the train across the corresponding track. Furthermore, it is possible to consider multitasking with multiple trains, one for each execution path.

Although it is clear that both the program and control flow and the train can be represented by an underlying graph which could in many cases be identical, it is nevertheless useful to look in greater detail at the recommended systems they would represent. In the following sections the various types of program control constructs will be examined.

7.1.4.2 Sequential Code Block

The most fundamental code structure is that of a sequential stream of instructions. This can be represented on a train layout as a section of track with no switches or crossovers, as shown in Figure 15. Photocells at various points along the track can be used to represent either individual statements or blocks of statements, i.e., when the train reaches a particular photocell, completion of the corresponding code block is indicated.

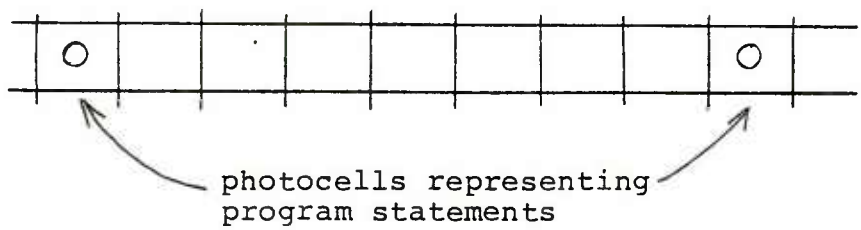


Figure 15. Representation of Sequential Instructions

7.1.4.3 Do Loops

There are several forms of do loops used in programming languages. The oldest is the do with iteration count. More recently do until and do while loops have become prevalent. Figure 16 illustrates a possible correspondence between loop program structures and the train layout. Figure 16 A illustrates a do until loop. Note that the code block is always executed

once. Figure 16 B illustrates a DO WHILE loop. Notice that in this case the code block is in the return loop and may or may not be executed depending upon the test condition performed before entry to the switch configuration. A do with iteration count could be represented by the same layout as a do until loop.

Figure 16 C represents a multi-purpose loop block which could represent either a do until or a do while loop depending upon where one presumes code of the loop to be represented. Photocells represent statements in both the forward and return loop. It would be possible to represent each variety of loop as being considered by using auxiliary light on the layout to indicate the active (those that refer to actual facts of code) photocells.

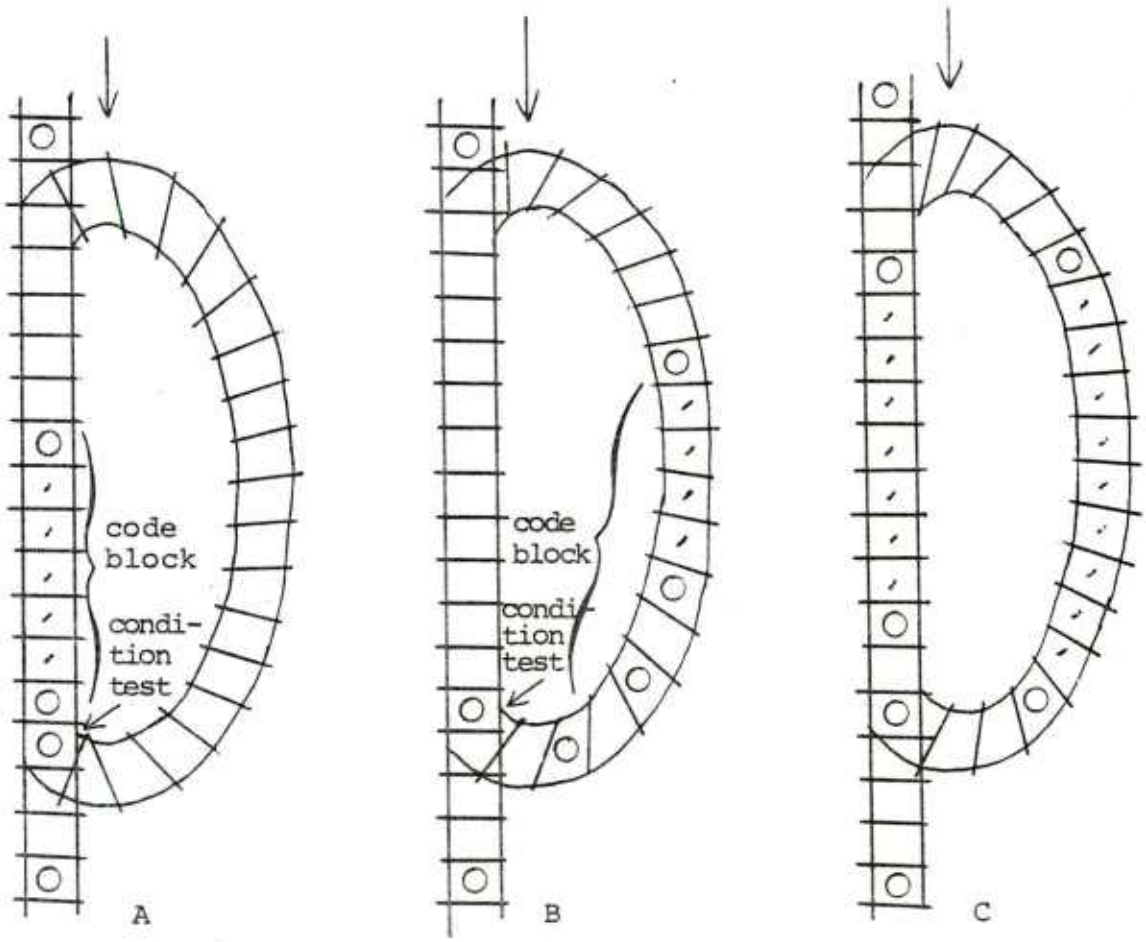


Figure 16. Correspondence Between Do Loop and Train Layout

7.1.4.4 If...Then...Else

The If...Then...Else structure can be handled with switches in a manner similar to that used for loops. Figure 17 A illustrates an If...Then operation while Figure 17B represents an If...Then...Else. It is presumed that the condition is tested prior to reaching the switch. Depending upon the result of the condition the switch can be thrown either straight or turned resulting in either the Then cause being done if the condition was true, or bypassed if the condition was false. The only distinction between the If...Then and the If...Then...Else in this representation is whether or not there is a block of code for the Else clause. This can be generalized as was the do loop shown in Figure 17.

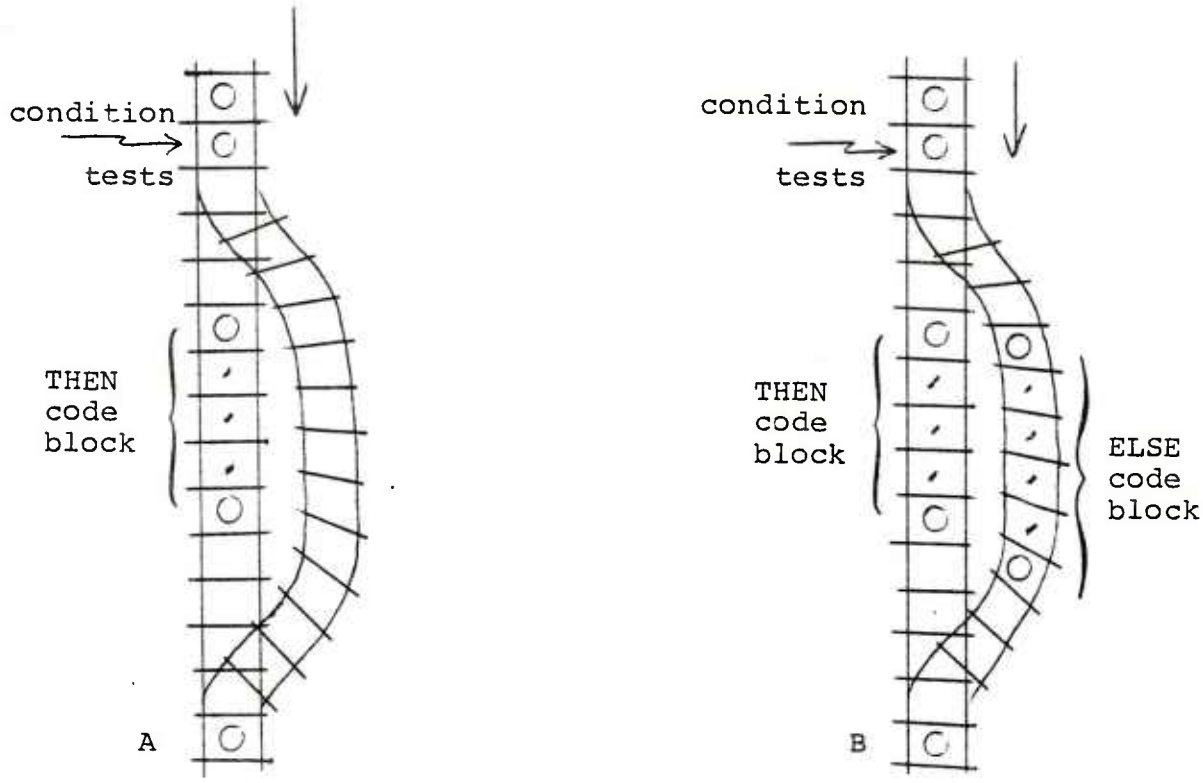


Figure 17. Correspondence Between If...Then...Else Clause and Train Layout

7.1.4.5 Go To

The Go To construct is simply a transfer from one stream of operations into another. It is represented simply as an inbound branch to another section of the track. This is illustrated in Figure 18 A.

A computed Go To is similar except that in the main stream of code one encounters a sequence of switches each of which branches to an inbound switch in another section of the track layout. This is illustrated in Figure 18 B.

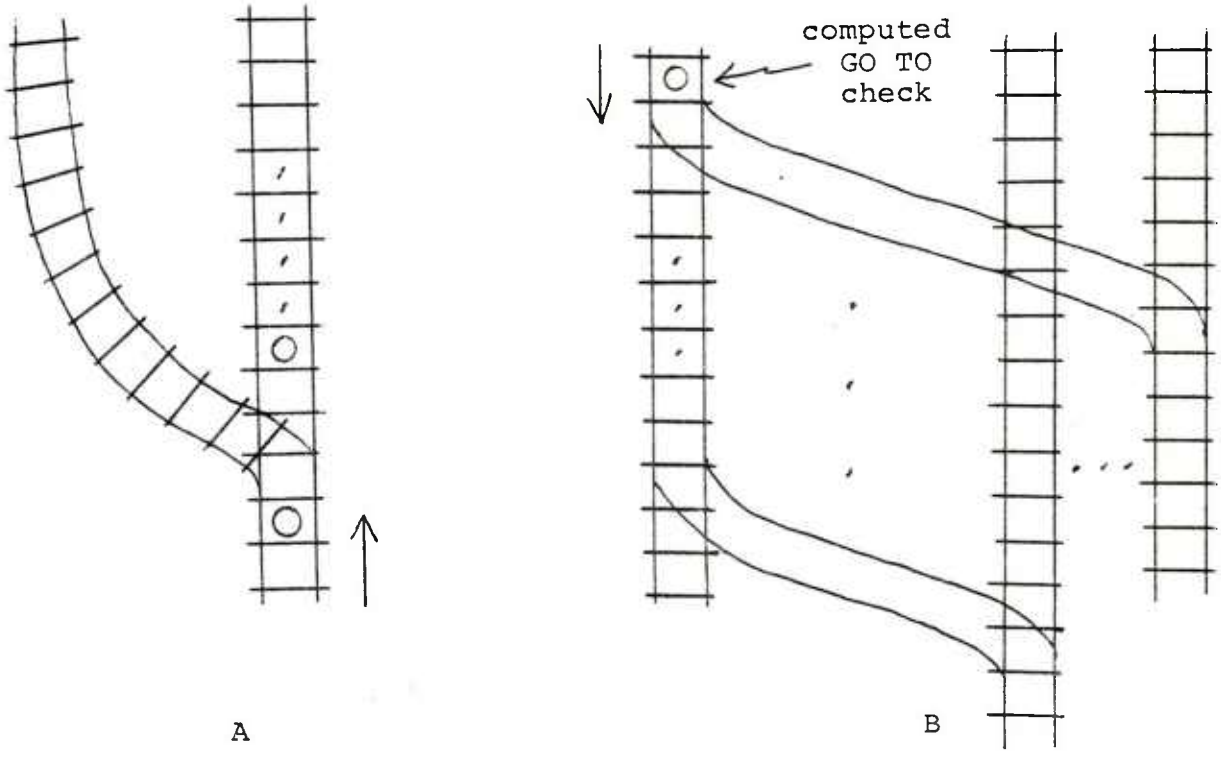


Figure 18. Correspondence Between GOTOs and Train Layout

7.1.4.6 Procedure Calls

A procedure call is basically a transfer to another block of code with a return upon completion of that block of code. The critical component here is that the procedure may be entered from several different places with an appropriate return being made. A representation of this is illustrated in Figure 19. The entry to the subroutine is simply a series of inbound switches to a block of code with their return being a series of outbound switches. The control computer must of course keep track of the

appropriate inbound and outbound switches so that a proper return is made. This is basically no different than managing subroutine returned within a procedure.

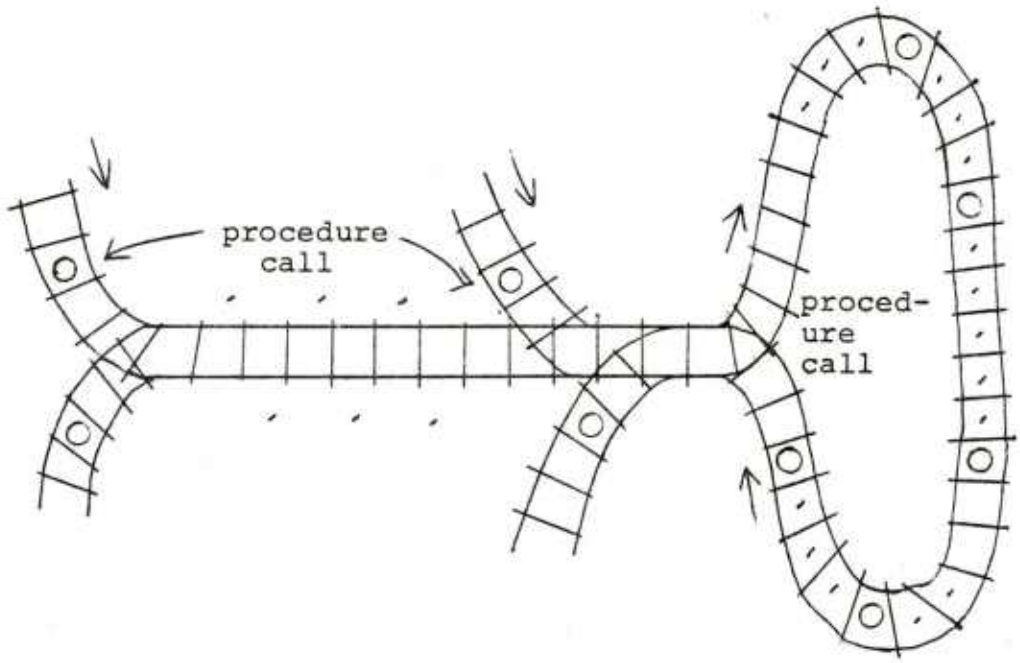


Figure 19. Correspodence Between Procedure Calls and Train Layout

7.1.4.7 Interrupts

The handling of interrupts and execution of a new task of code is managed by using multiple engines on the layout. Essentially one engine is present for each task. When an interrupt occurs the running train is stopped and a new engine corresponding to the new task is started. When the second engine either completes its traversal through the appropriate portion of the train layout (or is timed out) it is stopped and the original engine begins operation again. This is illustrated statically in Figure 20.

7.1.4.8 Operations

While there appears to be reasonably logical correspondence between control flow in a program and a track layout, any correspondence between noncontrol operations and the train is far less apparent. In fact some operations may not be possible at all. How does one represent X=Y+3.6892? Nevertheless there are some operations which conceivably might be modeled by the

train system.

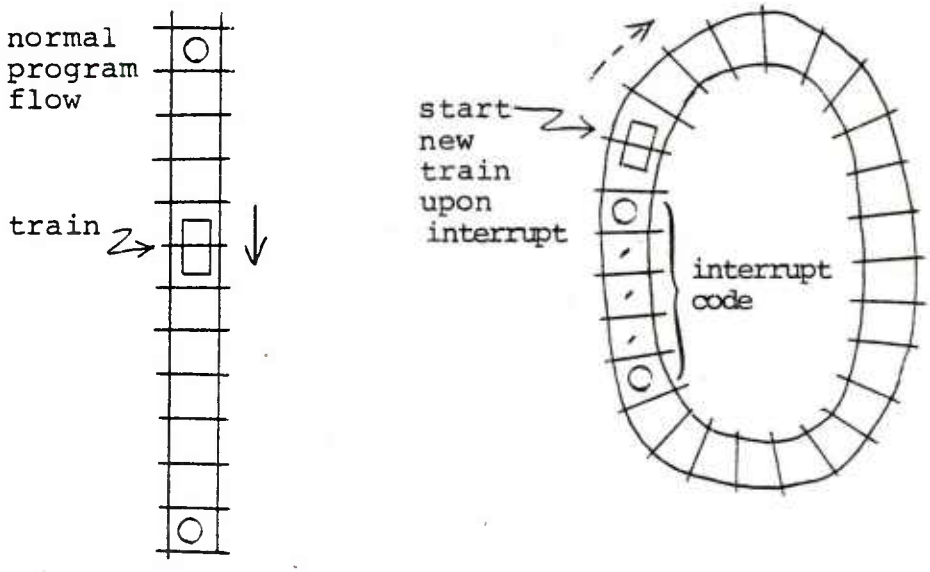


Figure 20. Correspondence Between Interrupts and Train Layout

One of the most fundamental types of operations one might wish to model is system I/O. In general, system I/O can be very complex and difficult to model. It does not seem reasonable to expect that it could be modeled in any quantifiable way. However, the occurrence of an I/O operation could be modeled in two ways. First, one might envision sidings with cars on them. An input operation could be modeled by backing into the yard and attaching an additional car to the train. Similarly an output operation could be the decoupling of a car from the train, with the car being left on a siding. Alternately one could consider an automatic load-unload facility for some item (perhaps small steel balls) from an open model car. In addition to not representing I/O in a quantitative manner, either of these alternatives is likely to be moderately difficult to implement.

An alternative would be to represent data types by model railroad cars. Different types of cars could represent different data types. While it does not appear feasible to have quantitative representation, one could use such a scheme to indicate all cases of data types within the procedure. Essentially the cars corresponding to the data types they represent would be picked up by the engine upon entering a procedure and would be deposited upon exit from a procedure.

This might be done only for those variables dynamically allocated. Again this is likely to lead to implementation difficulties.

Perhaps a more likely utilization of the train facilities for actual operation is in the representation and queuing operations. Many programs, particularly complex systems programs, involve substantial numbers of queuing operations. If this were done each queue would be represented by a string of model railroad cars parked on a suitable siding. FIFO and LIFO operations would be easily represented in terms of the addition of a car or removal of a car from the appropriate end of a section. Such an arrangement might help to visualize the effectiveness of various queuing algorithms.

7.1.5 Potential Program Train Coupling

The ideas presented in the previous section are associated with some potential areas in which the computer controlled train might be used to assist in visualization and validation of software. Implicit in each, however, is some connection between the actual software and the running of the train. It is basically assumed that the processor executes "in parallel" with the operation of the train and that the train is in fact a visual representation of what is going on within the processor. This carries with it several implications. First of all there must be a great slowdown of the executing program from its normal execution speed. Accordingly all appropriate timing input cues must be slowed down as well. Furthermore, and this is one of the most important difficulties, there must be a program connection between the executing program and the operation of the train. That is when a statement is executed the train must be told to move beyond that statement to the next. When a condition is evaluated and a branch is taken the appropriate switch must be set on the train layout and the train must be told to move over that switch. This section will discuss some possible ways in which these problems might be addressed.

7.1.5.1 Train Primitive

Before proceeding it should be pointed out that a set of primitive train operations can be written utilizing the techniques described in Chapter 3 to facilitate movement of the train in any of the schemes described below. It will be recalled that the hardware level control on the train implements two basic functions: the setting of a bank of eight switches to a set of desired positions and the setting of the power levels to individual electrically separate sections of track. It was noted that in order to operate the train effectively a moderately complex data structure and set of algorithms are required. Based upon these however a reasonable set of functions can be implemented.

For example, it is expected that the following functions would be reasonably straightforward to implement.

POWER (PVAL)	apply power VAL to track T
SWITCH (A,POS)	set switch A to position POS
FORWARD (E)	move train E in the forward direction
REVERSE (E)	move engine E in the reverse direction
TURN (S)	set switch S to the turn position
STRAIGHT (S)	set switch S to the straight position

Each of these functions can be implemented relatively easily in terms of the basic train system.

A very useful but basically more complex function would be a MOVE (A,B) which would be to move train A from its present position to photocell B. In general one would want the shortest path. For purposes of this application, however, the problem could be localized since calls of this nature would generally be made only to a relatively short distance from the present position of the engine. Accordingly the search space for the path could be greatly restricted to make the operation feasible. In fact a general path selection problem has been implemented on the train system at the University of Michigan.

A set of functions of this nature should be kept in mind when considering the discussions below.

7.1.5.2 Software Simulation

Perhaps the most promising use of the train facility for software validation lies in the area of software simulation. Moreover this is probably one of the easiest forms of utilization to implement. With this approach there is no formal direct link between the program and the simulation. Rather a separate simulation program utilizing the primitive train operations described above is written to correspond to program control flow. Program structures would be mapped into train movement utilizing the correspondences described in the previous section. As the simulator is executed the developer may then observe control flow by movement of the trains on the track layout.

This form of study is most likely to be useful during the design phase of the system. As noted earlier it is at this stage that the largest number and most difficult to correct errors are generally made. Assistance and validation at this stage may have greater impact on overall liability and cost effectiveness than elsewhere in the software life cycle.

7.1.5.3 Compiler Generated Calls

At the other extreme of program train coupling is the notion of having a compiler option which would insert the appropriate call to primitive train operations into the executable code stream as source program statements are processed. One would of course also need synchronization control; otherwise the program execution would rapidly run away from train movement. When the results of the program were executed it would result in essentially parallel operation of the program and the train visualization of the program. This is perhaps the tightest coupling one could expect.

However, this form of coupling is expected to be exceedingly difficult to realize. There are several problems with this approach. First is the matter of mapping a program structure onto the train layout. Though this is likely to be feasible to accomplish it is not expected to be easy. Furthermore no matter what algorithm is selected there always will exist programs which would not fit into a given layout. Finally, modifications of a compiler to this extent are likely to prove to be expensive, time consuming and difficult to debug in and of themselves.

7.1.5.4 User Inserted Calls

The idea of user insertion of appropriate train calls falls somewhere in between compiler generated train calls and a separate simulation program. This could be accomplished in two ways. First the user might insert the calls directly in the source code (perhaps with an appropriate tag in the comment field to allow easy removal after validation). This shifts the responsibility for mapping the program structure onto the track layout from an automatic procedure to the user where it is much more likely to be accomplished successfully. Furthermore it does not require extensive modification to program development software. It would, however, represent a significant increase in the programming load for the system developer.

The second method of user insertion of calls to the primitive train operations is through the use of a debug facility. Many debug facilities have the capability of insertion of subroutine calls at break points. That is, when a user defines a breakpoint he is also able to specify the address of a section of code to be executed when that breakpoint is reached. It is conceivable therefore that a user could set up correspondence between the program and the train facility at the time he was ready to execute the program through use of the debug facility. This would also allow him to look readily at only portions of the code, and control the degree of fineness with which he examines it (treating blocks of code as a single statement).

7.1.6 Limitations

All of these possible program train couplings have limitations of one sort or another. Limitations specific to a particular method have been mentioned in the section describing that method. This section will concentrate on limitations common to all of the possibilities.

One of the most obvious limitations is in the area of speed. Though a program can execute on even the smallest of today's computers at a rate of hundreds of thousands of instructions per second the train could not even handle a single instruction per second. This is most serious in the area of do loops. It is clearly not reasonable for the train to circle a loop any large number of times. Yet this may be necessary to represent what is happening within the control flow of a program. Perhaps a digital readout could be added to the loop and a single symbolic loop made with the train with the actual number of iterations displayed on the readout.

Technically there is a lack of representation of program operations. It is clear that some things can be modeled but there are certainly many which cannot. The extent to which meaningful validation can be carried out without a more complete model is still unknown.

Finally there is a matter of the configuration of the train layout itself. Large complex software systems also have a large number of control statements, many loops, many branching decisions, and many procedure calls. Representation of an entire system on a train facility would require a very large train facility. Moreover unless one designs a specific train layout for each program (a time consuming and costly operation) one would have to strive for a general train layout which could be mapped into a wide variety of program structures. This would require a means for some way of "blanking out" unused areas of the layout. This might be accomplished through the use of small signal lights along a layout. However, there is always the problem that no matter what track layout is present a program structure can be found which will not map into it.

The last issue can be ameliorated to some extent through the use of modularity in code design and in code validation. There is no need for each photocell on the track layout to represent a single station. Rather it could represent a block of statements or procedure which has previously been verified. In this way the size of the model might be reduced to one which would fit on a given train layout.

It is the feeling of these investigators that it is not really practical to try to represent complex software systems directly via a computer run train system and they are unwilling to endorse these ideas. If one were to proceed, however, it is

felt that the use of the facility for simulation at the design state or the use of simulation simply to provide visualization of how the program works, say for training operation, would represent the most likely areas for further investigation.

7.2 Data and Process Flow

The use of the train to model program control flow, was considered in the previous section. In this section a dual to that concept is considered, the use of the train to model data flow or some real process flow. In some respects the use of the train to model data or process flow may be more likely to yield useful results in its use for program control flow. First of all, this use of the train does not necessarily require as detailed a mapping to the train as the instruction mappings considered previously. Moreover, modeling process flow opens an application other than validation, namely operator training. The visualization provided by the train may provide a useful tool training operators in the use of the computer for complex process control applications.

7.2.1 Concept of Operation

The basic idea is to use the movement of the train to represent (i.e., model) some type of flow in the overall system being studied. It is the movement and position of the train or its associated railroad cars that are of interest.

One example of this discussed briefly in the previous section is the use of the train to model data flow in a program. While in general it is not reasonable to use the train to model operations on the data, in some circumstances, useful results can be obtained. In particular, if the operation involves a logical movement of data, a train representation might be possible. In particular, it might be used to model various types of cues. In Figure 21 below the siding could represent a queue. The cars on the siding represent items in the queue. Since the siding may be entered from the main track at either end, the siding could represent either FIFO or LIFO queue operations by adding or deleting cars from either end. Of course, one could have many such sidings to represent different queues and could use different types of cars to represent different types of items on those queues.

A second class of examples for which process flow modeling might be useful is for comupter controlled manufacturing operations. Such a system might consist of several individual assembly or processing stations with a flow of partially completed material through the various stations. Each station

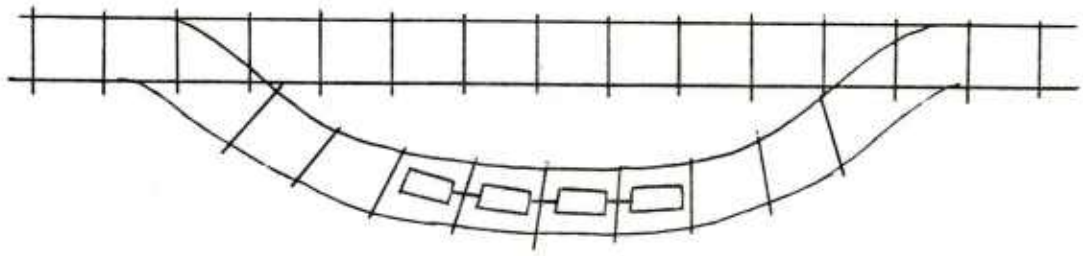


Figure 21.

would typically have material queues for incoming and outgoing materials. Such a system might be under the control of a single computer or a hierarchical system of computers with a processor for each station and a supervisor computer which monitors and controls the actions of the individual stations. A single such system is sketched in 22 below.

Use of the train system to model such a manufacturing

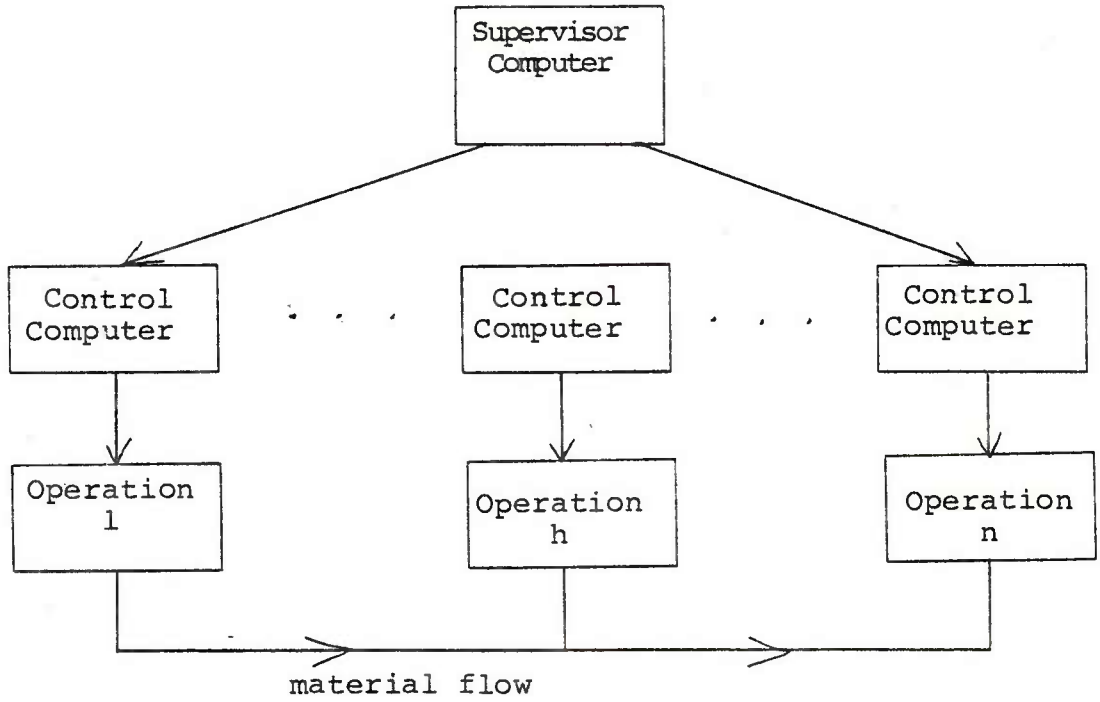


Figure 22.

environment might proceed as follows. The engines under computer control would represent the transport system between manufacturing stations. Open model railroad cars would represent

containers for material being processed at the various stations. These cars might contain small steel balls to represent the material itself. As in the data flow example above sidings could be used to represent material queues. Manufacturing operating stations might be represented by model load-unload stations to move the steel balls (magnetically) from one open car to another.

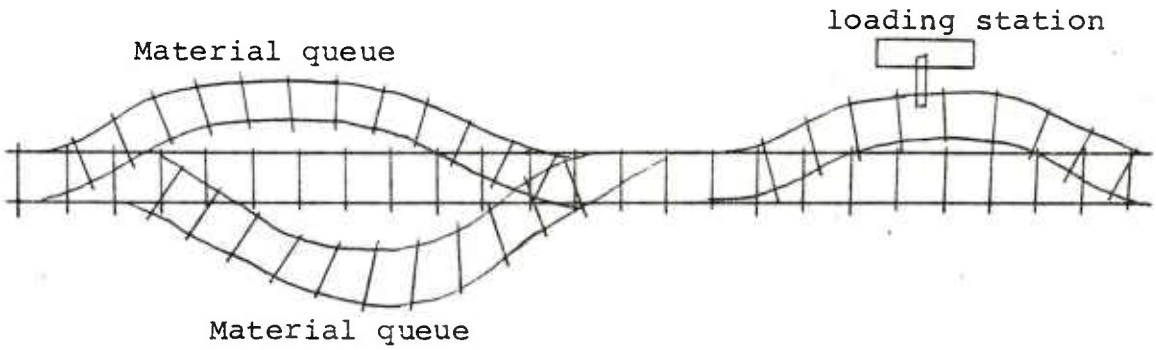


Figure 23.

The use of a computer controlled train facility to represent data or process flow would typically not be for software validation in the sense described above. Rather it might be used to study flow algorithms, e.g., queuing algorithms, in the early design stages. Visual representation afforded by the train might be useful during design.

Secondly, the study might be useful as a training tool even for operations which are not totally computerized. For example a shop foreman might be charged with the responsibility of managing the routing of materials from one manufacturing station to another. The visualization afforded by a train simulation might be useful in illustrating to him the real effects of various scheduling algorithms he might elect to use.

7.2.2 Implementation Considerations

There are basically two requirements which a process must meet before simulation by the train system could be considered. First, there must exist a mapping from appropriate process variables to the train facilities. This is highly application dependent and general rules are not easily stated. Two examples of such a mapping are cited above. In general movement or position of the train and/or model cars must be relatable to some significant parameters within the process being studied and the visual representation afforded by the train should assist either the designer or an operator in some way.

Secondly, the speed range of the train must be acceptable. That is, it is unreasonable to use the train to represent a data queuing operation which takes place at the rate of 100,000 data movements per second. To be useful in a situation like this it would be necessary that there be a meaningful shift in time scale which could be accomplished with only a small number of movement operations actually studied.

Given that the modeling of a process by the train system is in some way reasonable, there are two approaches to implementation which might be considered. The first is straight simulation. In this case there is no physical connection between the process being modeled and the representation in terms of the train system. Essentially one would write a separate simulation program to effect the desired movements on the part of the train. This is similar to the simulation approach described in the software validation section above. This is probably the easiest mechanism which could be used and particularly for training operations might well be quite satisfactory.

When the process under study is one which involves control of some external (to the computer system) real process an alternative implementation might be considered. For example, suppose that each station in Figure 22 above is computer controlled. One could consider removing the control output from the real devices to be controlled and replacing the devices with sensors that would be inputs to the computer controlled train facility. The train computer would then receive all control commands for the manufacturing devices through an internal mapping program which would have to be written. These commands could be translated into appropriate train movement commands. In addition the train computer facility would have to generate appropriate inputs to the manufacturing computers.

This alternative approach does in fact have some bearing on the software validation process as it provides an interaction with the actual control software in the manufacturing computers and presents a visual display of the operations they are performing. This approach, however, is likely to be more complex than the simulation approach in that one has to develop a number of hardware sensors as well as a suitable software program.

7.2.3 Discussion

As with software validation, the use of the computer train facility to model data or process flow requires that a suitably general track layout and auxiliary facility (e.g.,loading and unloading stations) exist. One again has the problem of either developing a special train facility for the process to be modeled or trying to develop a large general train facility to model a broad class of processes. The latter case suffers from the problem, of course, that there always exists processes which would not fit any given train layout.

Nevertheless the authors feel that it is in this type of application that the train facility offers the greatest promise. The authors are not yet willing to endorse strongly such a program based upon the limited study to date. Nevertheless, if suitable applications exist it is believed that this type of activity may be the most fruitful for future work.

7.3 Modeling Distributed Sensor Systems

This section will examine briefly the methodology for modeling the logical aspects of distributed sensor systems using the computer controlled model train system. Recall from Section 1.1 that our model of a sensor was a generator of data which could be activated either upon synchronous request of the processor or upon the occurrence of an external event. Recall also the desired property is an analog to be used for studying distributed sensors. From the foregoing discussions it is evident that the facilities of the real time computer applications laboratory satisfy these conditions.

The train system is itself a distributed sensor system with two categories of sensors, position sensors (the photocells) and value sensors (the throttle sensors). Normally these operate in asynchronous mode based upon external events. However since the controlling computer system has a real time clock, they may also be operated synchronously upon command from the computer. The multi-tasking capabilities of the system software developed for the computer make it possible to implement various queuing strategies for dealing with concurrent or nearly concurrent external events. Moreover, since the computers in the laboratory may be interconnected it is possible to emulate the effects at hierarchical levels of observer activity. Consequently, the computer controlled train system does meet the conditions set down for an analog of distributed sensors.

The first step in modeling sensor activity is to develop mapping between the distributed sensors and those available on the train facility. In most cases this implies a restriction of the set of sensor systems which may be considered. When one considers the manner in which interrupts from the photocell sensors on the train occur it is apparent that they do not in general occur at random. Normally they are the result of a sequence of interrupts from contiguous sensors on the train layout. Accordingly, the system is most appropriate for the modeling of sensor systems in which there is some expected movement through the sensor field. This restriction need not be totally applied, for there is no such restriction on the throttle sensors, however, since the largest number of sensors are photocells it would be a common restriction. This restriction could be somewhat ameliorated by using alternate mechanisms for causing photocell interrupts, e.g., passing some other object over them.

Once a mapping of sensors to the photocells has been completed, a clear definition of actions to be performed upon the occurrence of an interrupt must be defined. If these actions require supplemental input, the user must provide test values from a secondary input source, e.g., a disk file.

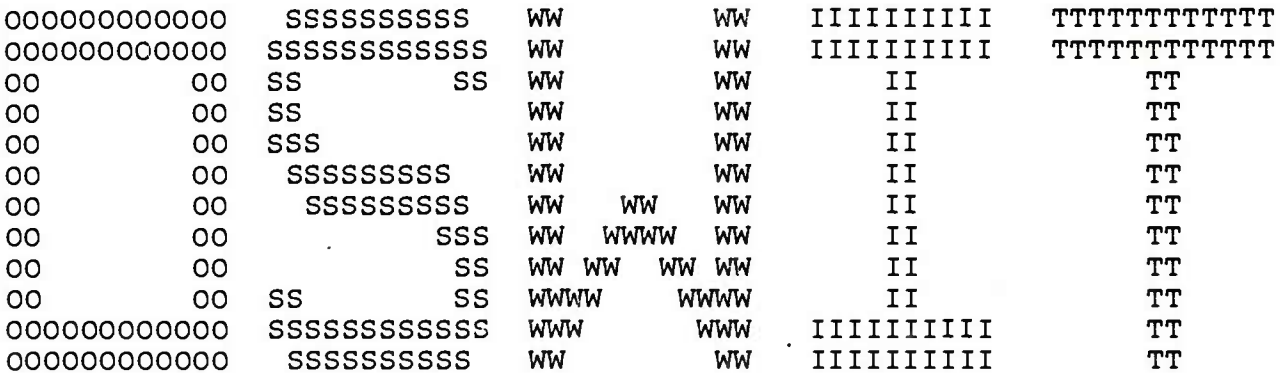
The actions being modeled must be implemented on the computer system. Most likely they will take the form of a recording and/or processing of the information obtained from the sensors. The interconnection between the computer and the laboratory allows for a study of hierarchical interconnection between processors. Generally the code at this level of simulation would be quite similar to that in the system being modeled. The train system would merely provide a visualization of the operations.

The generation of the interrupts would be controlled by a human observer/controller. Underlying the sensor-observer interconnections being modeled would be a separate set of tasks such as those described in Section 3.4 for controlling the movement of the train. Desired motion (i.e., sensor activity) could be generated either by a file describing the desired routing or by the human observer/controller using a subset of the train throttles.

It is also possible to add auxiliary controls and sensors to the model train system which would augment its modeling capability. Automatic coupler-decoupler systems for model railroad cars could be easily added. Moreover, to represent material of one sort or another, mechanisms for loading and unloading items (e.g., small steel balls from open cars could be added. Perhaps more usefully, it is possible that bar codes similar to those currently in wide use in grocery stores could be added to model railroad cars. Passing the car past a read mechanism would enable identification of that car. In other words, the class of sensors which could be represented by the system system would be increased.

With procedures such as this, it is possible to represent a reasonable class of distributed sensor systems observe behavior under varying inputs.

APPENDIX A



Operating System WIth Trains

U S E R S

M A N U A L

August, 1979

Richard A. Volz
Richard M. Jungclas

TABLE OF CONTENTS*

*Page numbers in the Table of Contents refer to the numbers in the upper corner of each page.

1. OSWIT: AN OVERVIEW

1.1 Introduction

The field of digital computers and their applications is perhaps the most dynamic field in engineering at the present time. Driving this change during the past ten years has been the introduction and widespread acceptance of the microcomputer. There are numerous products on the market using microcomputers, and the future is almost limitless. At present, however, software support for these systems lags far behind their older and larger counterparts. The availability of microcomputer operating systems is rather limited. Most present microcomputer operating systems are not really suited to real time applications that are forthcoming for microcomputers. During the next decade it is important that suitable real time operating systems be afforded the developer of microcomputer applications.

OSWIT (Operating System WIth Trains) is an operating system developed at the University of Michigan to meet real time executive system needs for the Digital Equipment Corporation LSI-11 microcomputer. The basic features of the operating system were designed and implemented by Jack Bonn and Ted Kowalski as an independent study project under the direction of Professor Richard A. Volz in the fall of 1975 and winter of 1976. During the fall 1976, Bill Dargel was responsible for the design and implementation of the disk controller. In addition, Kent Hoult developed and implemented the file system while Arnold Vance implemented the A/D and D/A drivers and train interface. In fall of 1977, Houton Aghill completed the design and installation of the MCP protocol between OSWIT and MTS. In fall 1977, Kent Hoult continued the development of OSWIT and the file utilities. Carol Briggs, Mark Hanie and Glen Purdy later modified the I/O structure to allow transmission rates of 2400 baud. Rick Richardson modified OSWIT to support DEC compatible soft sectored floppy disks at other locations within the University.

The basic features of the OSWIT operating system are:

1. A task scheduler which functions with a programmable clock and asynchronous events to start task by various methods subject to a specified software priority.

2. A wait structure to allow processing and I/O operations to proceed in parallel.

3. Input/Output device drivers for the console, A/Ds, D/As, floppy disk, paper tape reader, and printer.

4. MCP protocol to allow the microcomputer system to communicate with the University's central computer system (MTS).

5. A simple command structure modelled after the Michigan

Terminal System (MTS).

6. Floppy disk file system.

7. A small set of utility routines to support arithmetic conversions.

8. An absolute loader.

A brief overview of these features will be given here. They are described in greater detail in the OSWIT user's manual, presented in Appendix B.

1.2 OSWIT command language

The OSWIT command language provides the mechanism for user communication with OSWIT. This command language is modeled after the Michigan Terminal System command language. This command language permits system control, program control, a debugging monitor, file handling and communication with MTS.

This command language also supports logical unit assignment and pseudo device names similar to those used on MTS. Assignment of the physical devices to logical units may be done when program execution is initiated from the OSWIT command language or from wihtin an executing program.

Pseudo device names are used by OSWIT command language to symbollically refer to physical file or devices when the actual file or device names or address are not available. Pseudo devices names are provided for terminal output and input, paper tape reader, the line printer, the A/D and D/A converters, the train interface and a dummy file or device.

1.3 OSWIT file system and utility programs

OSWIT has mechanisms for creating, destroying, renaming, emptying, truncating, editing and cataloging disk files. To minimize the operating system memory requirements, these mechanisms are provided by an OSWIT utility program named *FILES11. OSWIT defines a file as a sequence of logical records placed in non-contigious, 512 byte blocks on the disk. A file cannot exceed 255 blocks.

Filenames are limited to 10 characters or less and may consist of any combination of printable, uppercase characters. Any filename starting with an "*" is designated as a public file and is usually reserved for OSWIT system files and utility programs.

OSWIT provides no file protection mechanism in the its catalog. The only protection of files available is through the WRITE ENABLE/PROTECT hardware switch.

In addition, other utility programs, such as *EDIT, *FILESNIFF, and *TIME provide additional user support.

1.4 OSWIT support functions

Internal to OSWIT are a number of support routines used by the operating system to implement its functions. These include numerical conversions, dynamic buffer management, I/O operations, task scheduling, etc, . As a general principle, all such functions are made available to user's programs. These functions are called at the assembly language level via emulator trap instructions (EMTs).

1.5 MTS - OSWIT communications

OSWIT uses the MCP protocol {1} to communicate with MTS on an Amdahl 470/V7. Each system is hardwired via a 1200-2400 baud line to a remote data concentrator, which statistically multiplexes the line with other units and communicates with MTS through a hardwired 9600 baud line. This mechanism is used principally to transfer data and programs between MTS and the local floppy disk, or to use the system as an "intelligent terminal". Source editing can be done locally, transferred to MTS for assembly or compilation, linked and the object file down loaded to be stored and executed on microcomputer system. Alternatively all development of user programs can be accomplished on MTS with the final object stored and executed locally. In addition, data may be collected and transfered to MTS for greater storage capacity or more thorough analysis.

1.6 Real time operations

According to Martin, {2} a real time computer system is one which accepts inputs from one or more sources, acts upon these inputs, and produces corresponding outputs fast enough to effect the source. This definition encompasses a wide variety of systems such as the use of a computer as a data concentrator, as the control element in a feedback loop, as a data logger for some real time process, or as a supervisor for a set of other real time computers.

There are two primary characteristics which distinguish real time application from the scientific computations: the need to respond rapidly to the occurrence of events external to the computer, and the need to handle I/O for a potentially large

1 UM Computing Center, "An MTS Communications Protocol (MCP) Proposal", May 1976.

2 Martin, James, Design of Real Time Computer Systems, Prentice Hall.

number of external devices in a manner which does not lock up the CPU during the I/O transfer. An example would be to require a computer controlling electric power distribution to suspend normal program operations upon detection of a generator failure and initiate an orderly shutdown procedure for that generator and a redistribution of the load among the remaining generators. The consequences of these characteristics are far reaching.

1.6.1 Tasking

First, in order to allow the user to specify the response to external events, he/she must be given some control over interrupt handling. Secondly, since the computer is usually much faster than the devices it controls or responds to, it is common to have a single computer control a number external devices. As a result, one usually has several more or less independent pieces of code known as tasks which are executed at different times. OSWIT provides a mechanism for associating a task with an interrupt or a condition for a given external device. When an interrupt occurs the program currently operating may be suspended and the associated task executed. When this task is completed, its execution is terminated and the original program is resumed.

Associated with notion of task is that of a priority. If two or more tasks are competing for the CPU, there must be some mechanism for deciding which task is to execute. In OSWIT each task is assigned a priority. Once started a task will run to completion unless interrupted by a task with a higher priority. If task A has priority of 10 and is interrupted by task B with a priority of 25, task B will execute until completion unless interrupted by a task with a priority greater than 25. When task B finishes, task A will resume.

1.6.2 I/O and interrupt structure

The OSWIT I/O and interrupt structure is generalized and oriented toward real time applications. All I/O operations at the programming level are done through logical unit assignments. Assignment of physical devices to logical units may be done at the time program execution is initiated or dynamically from within the program. All I/O requests to OSWIT do an immediate return to the calling program after the request is initiated so that processing may be overlapped with I/O. If an I/O operation must be completed before the task can proceed, the task may issue a WAIT request to OSWIT.

OSWIT supports logical record (line), byte, word and character I/O. OSWIT also supports requests for decimal or octal character string to binary word and binary word to decimal or octal character string conversion requests.

OSWIT supports tasks that require synchronous timing. The LSI-11 microcomputer hardware has a programmable real time

clock. The user can request OSWIT to set up time intervals in the clock and interrupt the CPU when the inverval has passed. This OSWIT facility allows the user to specify that a task is to be executed repeatedly at fixed intervals of time, at a certain time of day or after some interval of time.

2. SYSTEM OPERATING INSTRUCTIONS

Each laboratory station consists of a number of individual pieces of hardware. Their use as a coordinated system is controlled by the OSWIT operating system. This chapter identifies briefly the individual components of the system and describes how to operate the system, which includes the operation of the system hardware.

2.1 System hardware

Figure 1 shows the pictorial view of the system hardware. The components of interest in system operation are:

A. HALT switch
B. Disk WRITE PROTECT/ENABLE switch
C. Disk READ/WRITE LED
D. Hardware Bootstrap Switch
E. MAIN/AUX disk switch
F. Main disk drive
G. LSI-11 Backplane
H. Disk controller
I. A/D and D/A interface
J. D/A outputs
K. Digital display select switch
L. Digital display
M. Analog Display
N. Analog display select switch

The position of these components are indicated in Figure 1. The way each component is described in more detail in the following sections.

2.1.1 HALT switch

Depressing the HALT switch will place the system in HALT mode which in turn traps the system to the ODT (Online Debugging Tool) monitor. This monitor resides in a read only memory (ROM) on the CPU circuit board. It allows the user to (1) display and/or modify memory and register contents, (2) begin execution at a specified address, or (3) load an absolute core image in a special form (usually just a more general loader).

2.1.2 Disk WRITE ENABLE/PROTECT switch

This switch controls write protection on the system floppy disk. When this switch is placed in the WRITE ENABLE position, writes to the disk will be completed. When the switch is in WRITE PROTECT position writes to the disk are not completed but an audible alarm (two bells) is sounded. Since this is the only protection mechanism for the floppy disk, this switch should be set on WRITE PROTECT at all times, except when the user needs to WRITE to the disk.

RI J
8/21/79

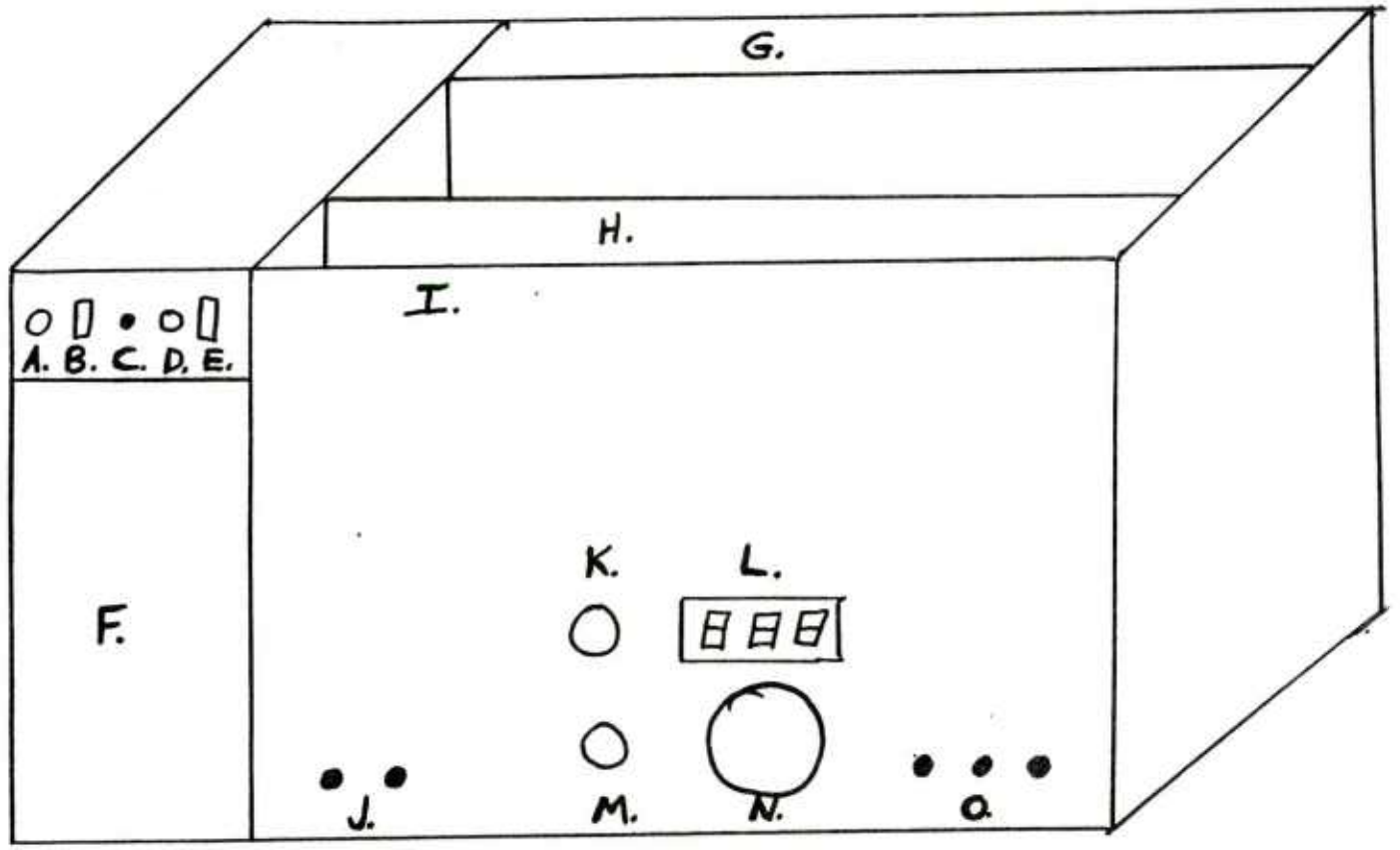


FIGURE 1: VIEW OF LABORATORY STATION

2.1.3 MAIN/AUXILLARY disk switch

Although there is normally only a single disk connected to a system, the disk controller is capable of controlling two disk drives. This switch is present for each disk drive present. When two disk drives are present, these switches selects which drive is designated as the main drive and which drive is used as the auxillary drive. With single drive units, the switch should be in the MAIN position.

2.1.4 Hardware bootstrap switch

The hardware bootstrap switch is used to bootstrap the operating system located in the file *OSWIT on the disk into core. Use of this switch is the normal way of loading OSWIT and beginning a session with the system. It is also often used to restore operation if the user's program causes the system to crash (fail). The operating system will not boot if this file is not present on the currently selected drive. The sequence of events is as follows. First, the hardware bootstrap loads and executes another bootstrap program residing in a file called *BOOTLOAD. This bootstrap program loads and executes an absolute loader program residing in a file called *ABSLOAD. The absolute loader loads and executes the operating system residing in the file named *OSWIT. Once OSWIT is initialized, the user will be prompted with an initial message.

2.1.5 Main disk drive

This is the standard disk drive present on all systems. It utilizes a hard sectored, single density diskette.

2.1.6 LSI-11 backplane

The LSI-11 is the heart of the system. This unit consists of a LSI-11 microprocessor, 56K bytes of memory, and serial and parallel I/O boards.

2.1.7 Disk controller

The standard disk controller interface will control up to two disk drives.

2.1.8 A/D and D/A interface

The A/D and D/A interface contains the A/D and D/A converters and other logic to interface the converter to the LSI-11. The standard interface will provide up to two converters of each type. to interface the converter to the LSI-11.

2.1.9 D/A outputs

These are the analog outputs from the D/A converters contained on the interface.

2.1.10 A/D inputs

These are analog inputs to A/D converts from external devices.

2.1.11 Digital display and select switch

This digital display will display the digital output from A/D converter or the digital input to the D/A converter. The display select switch will determine which device is displayed.

2.1.12 Analog display and select switch

The analog display will display the voltage input from the A/D converter or the voltage output from the D/A converter. The analog display select switch determines which device is displayed.

2.2 Loading OSWIT and fatal error recovery

2.2.1 Loading

The OSWIT operating system can be loaded or reinitialized by hardware bootstrapping or by running a bootstrapping program *BOOT. By pressing the BOOT button, the operating system residing in the file *OSWIT will be loaded, initialized and executed. The *BOOT utility program may be used to load, initialize and execute the operating system contained in any file. (This feature is typically used only by system programmers when working on a new version of OSWIT.) Once booted, the user will be prompted with '.', and the user will be in the OSWIT command language monitor. The OSWIT command language is described in detail in the next chapter.

2.2.2 Fatal error recovery

When a fatal error occurs, the program is unable to continue and will crash to either ODT with prompt of '@' or the DEBUG mode with a prompt of '*', depending upon the severity of the error and the system configuration. In some cases it is possible to recover operation without rebooting OSWIT.

From ODT, a cold start of the operating system can be performed by typing 140G. This is equivalent to starting with a fresh OSWIT load except that the presently loaded version of OSWIT is used. All I/O devices are disconnected from the user program and the program disappears. A warm start of the operating system may be performed by typing 144G. This attempts to leave the user program loaded with I/O devices connected. A

program restart is generally inadvisable. It is used prior to entering DEBUG mode to see what went wrong. When in ODT, the DECwriter must have the uppercase key down.

A return from the DEBUG package may be accomplished by entering an "OT" command.

2.3 OSWIT system memory configuration

Figure 2 shows the standard operating system memory configuration. The interrupt vectors are located from 0-377 (octal). The memory mapped I/O device status registers are located in last 8K bytes. OSWIT, the debug package, the system stack and buffer area are located in high memory as shown. The user program area always starts at 400 (octal). Approximately 18K words (36K bytes) of user program can be loaded.

The system can be reconfigured in two ways; by modifying the stack size, and by removing the debug package. The system stack size is dependent upon the number of nested subroutines and parameters stored on the stack. The default size of 1K should be sufficient for most applications.

By removing the debug package, the user recovers an additional 2K of memory space, which can be used for his/her program. When the debug package is removed the buffers and stack area are redefined.

2.4 MTS communications

OSWIT uses the MCP protocol to communicate with MTS. When the OSWIT MTS command is issued, the DECwriter is effectively connected to MTS and operates as a "standard" terminal, except that a control-Z is accepted as a control command to return control to the local operating system, OSWIT.

It is also possible to have program communication with MTS. The OSWIT pseudodevice *MTS* can be used by OSWIT commands, utility programs, or user programs to establish a data path directly between MTS and OSWIT as needed (see OSWIT "RUN" command in the next chapter).

One of the more useful features of the MTS communication is the transfer of files between MTS and the local floppy disk. There are many options, however, and care must be taken if the transfer is to be completed correctly. The following examples illustrate copying object and source files between MTS from OSWIT.

Example(s):

A. Binary files

1. From OSWIT to MTS

RMJ
8/29/79

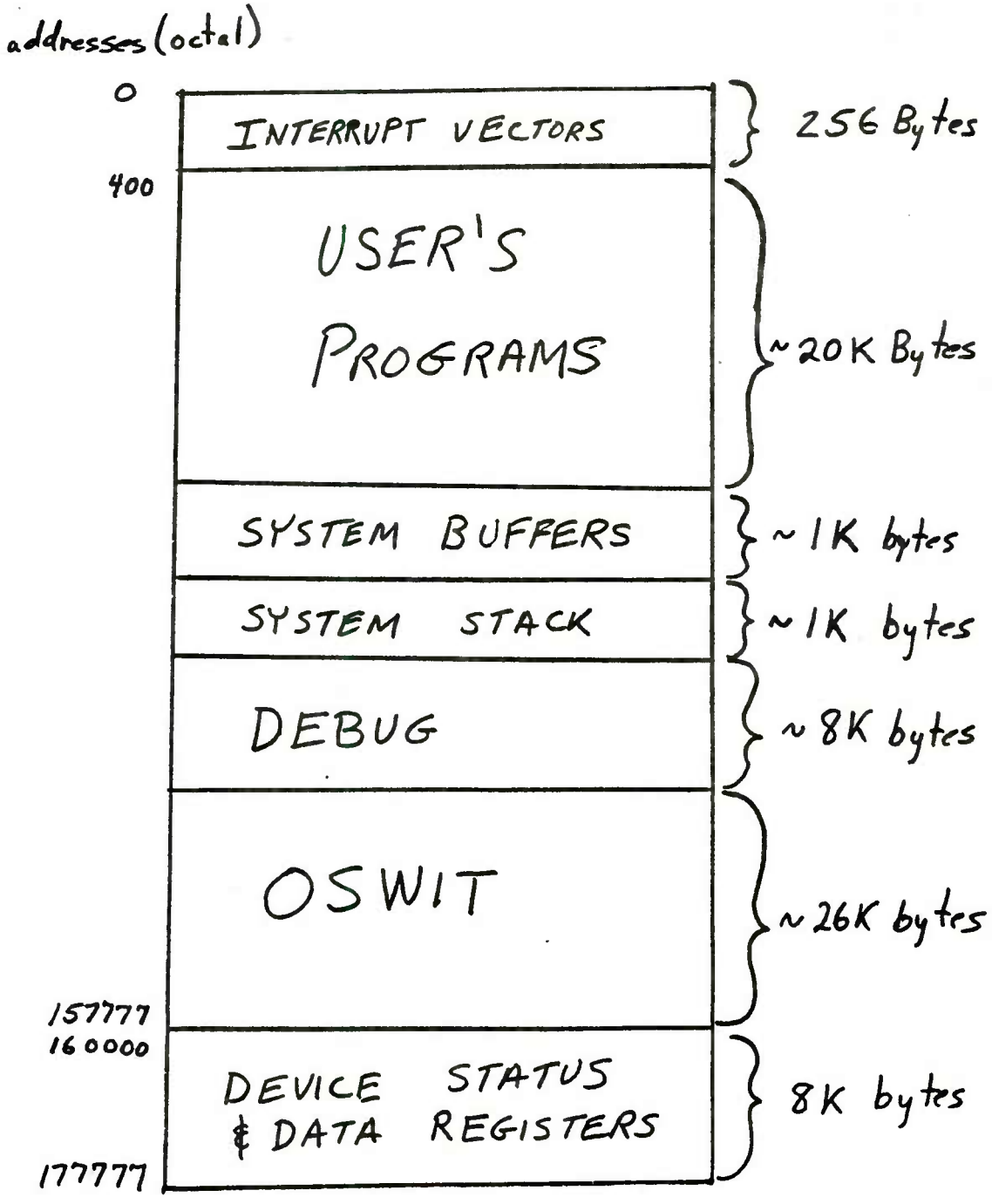


FIGURE 2: STANDARD OSWIT MEMORY CONFIGURATION

```
#COPY *SOURCE*@BIN MTSFILE
.COPY OSWITFILE *MTS*
```

2. To OSWIT from MTS

```
#COPY MTSFILE *SINK*@BIN@~TRIM
.COPY *MTS* OSWITFILE
```

where @BIN, and @~TRIM are MTS modifiers to transfer a binary file without character conversion and without trimming trailing blanks off the lines.

B. Source files

1. From OSWIT to MTS

```
#COPY *SOURCE*@SP@NCC MTSFILE
.COPY OSWITFILE *MTS*
```

2. To OSWIT from MTS

```
#COPY MTSFILE *SINK*@SP@NCC
.COPY *MTS* OSWIT FILE
```

where @SP is a MTS modifier that is a special I/O request to the data concentrator and @NCC is a MTS modifier specifies no carriage control is used and "" is a control-Z and the prompts typed by the active operating system are underlined.

3. OSWIT COMMAND LANGUAGE

3.1 Command overview

The OSWIT command language provides the mechanism for user communication with OSWIT. The command language permits system control, program control, a debugging monitor, file handling and communication with MTS.

All commands must start in column 1 and may be entered in either upper or lower case. Abbreviations consisting of a string of the initial characters of a command are allowed. The minimum abbreviations are underlined. The OSWIT command language interrupts always prompt the user with a ".".

The commands are listed below with a brief explanation of their function. A detailed explanation of each command is given in the following section, "OSWIT command language descriptions".

COMMAND SUMMARY

COMMENT [TEX1]

Insert comments on console listing

COPY [FDname1[FDname2]]

Copy contents of a file device to another file or device.

DEBUG

Enter DEBUG monitor

LOAD

Load a program without executing

MTS

Enter MTS through OSWIT

RESTART [AT location] [I/O FDname]

Restart or initiate program execution

RUN [object FDname] [I/O FDnames] [PAR=parameters]

Load and execute the program

SET parameter=state [parameter=state]

Change system parameters

START [AT location] [I/O FDnames]

Restart or initiate program execution

UNLOAD

Unload the currently loaded program

3.2 OSWIT command language descriptions

COMMENT

Command Description

Purpose: To allow insertion of comments on output to the terminal.

Prototype: COMMENT [TEXT]

Description: This command is ignored by the system, which allows the user to put comments in with his OSWIT commands.

Example(s): COM THIS IS A COMMENT COMMAND.

COPY

Command Description

Purpose: To copy the contents of a file or device to another file or device.

Prototype: COPY [FDname1 [FDname2]]

Description: Two FDnames may be given as parameters;

FDname1

FDname1 specifies the file or device that contains the lines to be copied (the input). If FDname1 and FDname2 are both omitted the input lines will be read from *SOURCE* and written on *SINK*.

FDname2

FDname2 specifies the file or device that is to receive the copied lines (the output). If FDname2 is omitted the output lines will be written on *SINK*.

The COPY command is a series of read and write operations. It causes lines to be read sequentially from FDname1 and written on FDname2 until the end of file is encountered on FDname1.

Example(s):

```
COPY A B
                File A is copied to file B

COPY A
                File A is copied to *SINK*

COPY
                *SOURCE* is copied to *SINK*
```

DEBUG

Command Description

Purpose: To allow the user to enter either ODT mode or to make a forced entry to the DEBUG package.

Protoype: DEBUG

Description: This command will issue an IOT instruction which will either be trapped, and an entry made to ODT via a HALT instruction; or to the DEBUG program, if it has been loaded.

To get back to OSWIT you may issue a "P" if in ODT, or "OT" (OSWIT) or "CE" (CONTINUE) if you are in the DEBUG program.

Example(s): DEBUG

LOAD

Command Description

Purpose: To load a program without initiating execution.

Prototype: LOAD [object FDname] [I/O FDnames] [PAR=parameters]

Description: "object FDname" specifies the file or device containing the program to be loaded. If omitted, the program is loaded from *SOURCE*.

The keyword parameters "I/O FDnames" are the assignments of logical I/O units to files or devices for use by the loaded program during execution.

The PAR keyword specifies an arbitrary string of characters to be passed to the loaded program on initiation of execution through an ADCON in R1.

Example(s): LO OBJECT 5=INPUT 6=OUTPUT PAR=RUN

This loads the program OBJECT. Logical units 5 & 6 are specified as INPUT and OUTPUT respectively and R1 is set up to point to a string in core which contains a 3 in the first byte (the length of the string) followed by the characters "R","U","N".

MTS

Command Description

Purpose: To allow the user to communicate with MTS through OSWIT.

Prototype: MTS

Description: The user should establish a link to MTS, then issue the MTS command to allow usage of the console device as a regular terminal connected to MTS. The link may be established though a telephone link or via a remote data concentrator(rdc) link. When using the rdc it is necessary to type an "extra" carriage return to establish the link from the rdc to the data concentrator. When using the phone link, it is necessary to select the full duplex and proper baud rate for correct communication.

The MTS mode may also be used to transfer data across the phone link. This is accomplished by referring to the pseudo device *MTS* in an OSWIT command. To initiate a transfer, MTS must start sending data, and OSWIT must start receiving it, or vice-versa. This is a two step process, where (1) MTS is told what to do, and (2) OSWIT is told what to do.

To stop the MTS mode the user types the special line terminating character: Control-Z (SUBstitute). While talking to MTS the Control-Z causes the current line to terminate. The line is then sent to MTS and control is returned to OSWIT.

Control may be returned to the MTS mode again at any time by re-issuing the MTS command.

The ATTN key always interrupts OSWIT and should not be used in for interrupting the MTS mode. A control-E is used in the MTS mode to attention interrupt MTS.

CAUTION: Avoid issuing multiple control-E's since these may hang the system (MCP protocol). Due to the high transmission rates, a large amount of transferred data may be present in OSWIT's buffers.

Example(s):

```
.    user is talking to OSWIT then
.    establishes a link to the Data  Concentrator
with
.MTS

#$SIGNON xxxx PW=xxxx
#$COPY MTSFILE *SINK*@BIN
.LOAD *MTS* SCARDS=*SOURCE* SPRINT=*SINK*
.MTS
#$SIGNOFF

.START
```

where the character in column 1 is the prompt, and the "" is a Control-Z.

RESTART

Command Description

Purpose: To restart or initiate execution of a program following either initial loading or an attention or program interrupt.

Prototype: RESTART [AT location] [I/O FDnames]

Description: The address at which execution is to begin is specified by LOCATION. The user can reassign the logical I/O units (see the RUN command description).

The restart command restarts (or initiates) execution of the currently loaded program. If a location is omitted the program is restarted at the point of the attention or program interruption or at the beginning if it was not already executing.

If logical I/O units have been reassigned, the files and devices originally assigned are CLOSEd and the newly assigned files and devices are OPENed.

Example(s): RESTART SPRINT=A

This restarts the currently loaded program with SPRINT reassigned to file A.

RES AT 20000

This command will restart the users program at location 20000.

RUN

Command Description

Purpose: To load and initiate execution of a program

Prototype: RUN [object FDname] [I/O FDnames] [PAR=parameters]

Descripton "object FDname" specifies the file or device containing the program to be loaded. If omitted, the program is loaded from *SOURCE*.

The keyword parameters "I/O FDnames" are the assignment of logical I/O units to files or devices for use by the loaded program.

The PAR keyword specifies an arbitrary string of characters to be passed to the loaded program on initiation of execution.

The run command calls upon the loader to load the object program into memory. If there are no fatal loading errors the comment " EXECUTION BEGINS " is printed and control is transferred to the entry point of the program by a standard subroutine call. R1 contains the address of a byte containing the length of the string passed followed by the actual string. If the program terminates execution by restoring the registers and doing a subroutine return the comment " EXECUTION TERMINATED " is printed.

All storage, files, and devices used for this RUN command are automatically released unless execution was not terminated normally.

Example(s): RUN LOAD

This loads and initiates execution of the program in the file LOAD.

R MYPROG 5=INPUT 6=OUTPUT PAR=QUIT

This loads and initiates execution of the program in MYPROG. Logical I/O units 5 & 6 are assigned to the files INPUT and OUTPUT respectively. R1 contains a pointer to a byte which contains the length of the string (4) followed by the letters "Q","U","I","T".

SET

Command Description

Purpose: To allow changing of various system parameters.

Protoype: SET PARAMETER=STATE [PARAMETER=STATE]

Description: The PARAMETER will be set to the specified STATE. If an illegal parameter or state is entered an error message will be printed and the rest of the line will be ignored. However all legal assignments before the point of the error will be set.

The legal parameters and states are as follows:

DEBUG=OFF

This will remove the DEBUG monitor from the system to give extra space for user programs. When removed, OSWIT will fatally trap memory and illegal instruction errors which were previously trapped by the DEBUG monitor. Using the OSWIT DEBUG command with the DEBUG monitor removed, will cause an entry to ODT. Typing P will restart OSWIT. The only way to get the debugger back is to reboot the system.

Default - Debugger is in system after loading.

STACK=SIZE

This will change the stack space available to the program. SIZE is the number of bytes to be allocated for the stack. It must be an even decimal number in the range 0 to the amount of memory available to the user.

Default - STACK=1024

Note: Setting the stack too large or too small could result in a system crash.

WV={ON/OFF}

To enable or disable verification of writes to the floppy disk. If write verification is

turned on then after every write the data will be read back in and compared with the original. If a write error has occured then the write will be retried up to 3 times. With write verification off disk writes will be faster and require less buffer space, but have an increased chance of error.

Default - WV=ON

Example(s):

```
SET STACK=1000
SET WV=OFF
SET STACK=500 WV=ON DEBUG=OFF
```

START

Command Description

Purpose: To restart (or initiate) execution of a program following either initial loading, or an attention or program interrupt.

Prototype: START [AT location] [I/O FDnames]

Description: This command is identical to the RESTART command.

UNLOAD

Command Description

Purpose: To unload the currently loaded program.

Prototype: UNLOAD

Description: The UNLOAD command unloads the current program in memory previously LOADed by the load command, or a RUN command if execution did not terminate normally. All storage allocated to the program is released, and all files and devices OPENed at execution time are released.

Example(s): UNL

4. OSWIT I/O AND INTERRUPT STRUCTURE

The OSWIT I/O and interrupt structure is generalized and oriented toward real time applications. All I/O operations at the programming level are done through logical unit assignments. Assignment of physical devices to logical units may be done either at the time program execution is begun or dynamically from the program itself (e.g., in response to user input). Further, all I/O requests do an immediate return to the calling program after the request is initiated so that processing may be overlapped with I/O. Both record and character user I/O are supported. When the user requires that an I/O operation be completed before proceeding, he must issue a WAIT call. If the physical device is in use when input request is made, an automatic WAIT is done, otherwise the request is queued. The user task may be started upon completion of an I/Ooperation.

A WAIT is accomplished with a WAIT EMT from an assembly language program and with a WAIT statement in a CRASH program.

4.1 System I/O directives

These system I/O directives represent supervisor "calls" which transfer information to OSWIT to perform predefined I/O functions. A brief overview of the I/O capabilti is presented here. A detailed description of the I/O directives may be found in appendix B.

4.1.1 Read and write operations

The I/O structure supports four types of READ and WRITE operations to logical units: byte oriented, character oriented, word oriented and logical record (line) oriented. These functions are performed from user's programs through EMT calls in assembly programs or CRASH statements in CRASH programs which are translated into EMT calls by the CRASH compiler.

4.1.2 Conversions

The I/O structure also permits character string to binary word and binary word to character string conversions. These functions are performed for octal and decimal character strings from user's program via EMT calls. For CRASH programs, number-string conversions are also provided for real numbers. In CRASH many of the conversions take place implicitly as needed (see CRASH manual).

4.2 Logical units

When a program is coded, the names of the files and devices to be used for input and output are normally unknown. Even if the names were known it would be inconvenient to specify them in the program, since this would require retranslation each time a file or device name is changed. Thus, it is desirable to specify

the names at execution time rather than at translation time. This is accomplished with logical I/O units. A logical I/O unit is a symbolic name used in a program to specify the source of data for input and the destination of the output information. A logical I/O unit does not identify a specific file or device. The logical I/O unit is used by the program as a reference. When the program is executed, each logical unit used by the program must be attached to the actual file or device. This is normally done on RUN command by specifying a keyword of the form of

unit=FDname

Alternatively, logical I/O units can be attached by calling the appropriate subroutine in CRASH or EMT in assembly language.

The logical units are numbered 0 through 30. Some of these logical unit numbers have specific character representations. These are:

Name	LUN
SCARDS	26
GUSER	27
SPRINT	28
SPUNCH	29
SERCOM	30

These character representations are equivalent to the MTS definitions.

Since it is desirable to minimize the information needed on the RUN command, some of logical I/O units have default specifications. The following defaults are provided if no logical I/O unit assignment is given with the RUN command:

Name	Default
SCARDS	*SOURCE*
SPRINT	*SINK*
SPUNCH	*SINK*
SERCOM	*MSINK*
GUSER	*MSOURCE*
0-25	*SOURCE* or *SINK* (Depending on usage)

4.3 Pseudodevices

4.3.1 Psuedodevice overview

A pseudo device name is used to refer to a file or device if the actual name is not available. Pseudo device names are needed for the paper tape reader, terminal input and output, the line printer, the A/D and D/A converters, the train interface and dummy devices or files. These pseudo devices are predefined and described in the next section.

Pseudo device names begin and end with an asterisk. The characters in name may be entered as uppercase or lowercase. These devices can be connected to any logical unit.

Example(s):

```
.RUN   MYPROGRAM   SCARDS=MYFILE   SPRINT=*SPRINT*
 0=*CO*
```

4.3.2 Pseudodevice descriptions

CONVERTER0 and *CONVERTER1*

Pseudodevice Descriptions

Purpose: To allow access to the D/A and A/D converters.

Ref by EMT(s): READB

WRITEB

Description: These pseudodevices allow users to assign units to the A/D and D/A converters.

A READB from this pseudodevice will result in the current A/D value being input. A WRITEB will cause an output to the D/A. The trailing number of the pseudodevice name corresponds to the A/D, D/A set which it controls. These pseudodevices cannot be reassigned.

Example(s): RUN SERVOPROG 2=*C0*

DUMMY

Pseudodevice Description

Purpose: To allow access to the infinite wastebasket(for writes) or empty file(for reads).

Ref by EMT(s): All I/O

Description: *DUMMY* is a null device which will accept output lines and do nothing. On input, it returns with an endfile condition. The pseudodevice is useful for routing unwanted output.

MSINK

Pseudodevice Description

Purpose: To allow write access to the master sink device(terminal).

Ref by EMT(s): WRITEB

WRITE

Description: *MSINK* is similar to *SINK* except that:

1) It is always assigned to the terminal

2) It cannot be reassigned

MSOURCE

Pseudodevice Description

Purpose: To allow read access to the master source device(terminal).

Ref by EMT(s): READB

READ

Description: *MSOURCE* is similar to *SOURCE* except that:

1) It is always assigned to the terminal

2) It cannot be reassigned

MTS

Pseudodevice Description

Purpose: To allow access to MTS

Ref by EMT(s): READB

READ

WRITEB

WRITE

Description: *MTS* allows user programs to communicate with MTS via the modems connected to the LSI-11s. This pseudodevice cannot be reassigned.

PRINT

Pseudodevice Description

Purpose: To allow access to the line printer.

Ref by EMT(s): WRITE

Description: Writing a record to *PRINT* will cause the line printer to print the record, using the first character for carriage control. Valid carriage control characters are (+,-,9,1, ,0) and have the same meaning as with MTS.

Example(s): COPY PROGLIST *PRINT*

READER

Pseudodevice Description

Purpose: To allow access to the paper tape reader.

Ref by EMT(s): READB

Description: Assigning *READER* to a unit number will result in the paper tape device being referenced. A READB will result in a byte being read.

Example(s): RUN TAPEREAD 0=*R*

SINK

Pseudodevice Description

Purpose: To allow access to the current sink file or device.

Ref by EMT(s): Depends on assignment.

Description: Accessing *SINK* will reference the file or device currently assigned to it. *SINK* is assigned initially to *MSINK*. Any write operation by an unassigned unit will default to *SINK*.

Example(s): RUN PROG2 1=*SINK*

Output written to unit 1 will reference the file or device currently assigned to *SINK*.

SOURCE

Pseudodevice Description

Purpose: To allow access to the current source file or device.

Ref by EMT(s): Depends on assignment.

Description: Accessing *SOURCE* will reference the file or device currently assigned to it. Initially, *SOURCE* is assigned to *MSOURCE*. Any read operation by an unassigned unit will default to *SOURCE*.

Example(s): RUN PROG 0=*SOURCE*

When PROG references unit 0, it will reference the file or device assigned to *SOURCE*.

TRAIN

Pseudodevice Description

Purpose: To allow access to the devices associated with the train.

Ref by EMT(s): READW

Each word read from the train interface is passed directly to the user program. The high order bit of the word is set for a photocell interrupt and reset for a throttle interrupt.

WRITEW

The word passed to WRITEW is written directly to the train interface.

Description: This pseudodevice allows users to assign units to the train devices which consist of:

1) Switches

2) Tracks

3) Photocells

4) Throttles

This pseudodevice cannot be reassigned. See READ, WRITE, READW, and WRITEW EMT descriptions for the calling procedures.

Example(s): RUN TRAINPROG 6=*TN*

5. TASKING AND TIMING

5.1 Tasking and Timing introduction

It is sometimes desirable to have many different activities take place concurrently within the computer. Normal subroutine calls cause suspension of the calling program until the called program has returned. OSWIT allows several procedures to be active at one time, without requiring the completion of one before another can execute. Procedures which can "live" independently of other procedures are called tasks.

There is a variety of ways a task can be scheduled to execute. It can be synchronized with the clock, with the procedure which first invoked it, or with some other procedure. It may even be scheduled to execute asynchronously (triggered by some external event or I/O completion).

Any task has both a scheduling attribute and a priority attribute. The priority attribute specifies the relative importance of each task in the collection procedures being executed. The basic tasking functions that are part of the OSWIT system directives are:

AT	Starts a task at a certain time of day
CANCEL	Cancels the task
DEFINE	Creates and defines a Task Control Block
DESTROY	Cancels and destroys the TCB
EVERY	Starts a task every interval of time
IN	Starts a task in an interval of time
LOCK	Locks the active task to prevent pre-empting
ON	Starts a task on the occurence of a condition
START	Starts task immediately
WHENA	Starts task when interrupt from channel A occurs
WHENB	Starts task when interrupt from channel B occurs
UNLOCK	Restores task's orginal status before LOCKing

The general meaning and use of these functions are described in the following sections. A detailed description of these EMTs can be found in Appendix B.

The CRASH tasking and scheduling statements are translated into these EMTs by the CRASH compiler.

5.2 Task definition

Each task occurrence must be known to the operating system. The DEFINE EMT creates a task control block (TCB) (a block which contains information about the task, e.g., its starting address, priority, time interval, etc.) and enters it in a DEFINED queue. Up to 255 task control blocks may be allocated by OSWIT before additional requests are ignored. A single piece of code may be used by multiple tasks (multiple TCBs required) when it is

desired to use the code for more than a single type of event occurrence. For example, a piece of code may be scheduled for either of two types of asynchronous interrupts. Three operations pertain to usage of tasks; (1) Task definition, (2) Task scheduling and (3) task invocation. Task definition specifies to OSWIT the existence and the attribute of a section of code that is going to be used as a task. Task scheduling causes the task to be entered into a scheduled queue where the task awaits for some event that triggers the task to began execution. Task invocation occurs, when some event causes the task to be entered into the execution queue. Task scheduling and invocation may occur at the same time or at distinct times. For example, START schedules and invokes the task concurently, while scheduling and invocation for ON and EVERY occurs at distinctive times.

5.2.1 Task identifiers

Since the same piece of code may be defined to execute for several different events, just giving the name or the address does not uniquely specify which invocation of the task is meant when it is referred to. Therefore, whenever a task is scheduled, the scheduler returns a unique identifier for the particular definition. Subsequently, the task is identified by its task identifier.

5.2.2 Priority

An invoked task is entered into the collection of tasks competing for execution. Since only one task may proceed at any time the one selected to proceed is the one with the highest priority. If a task with a higher priority is invoked, it preempts the lower priority task until it is completed. There is neither time slicing nor sharing in OSWIT by the scheduler during I/O. The highest priority task in competition for CPU time will run to completion before any lower priority task can proceed. Within a given priority level, tasks are started in the same order in which they are scheduled.

It is recommended that user tasks be restricted to priorities between 5 and 250. The other priorities are reserved for system use. The main task (program) has priority of 10.

5.3 Task scheduling

Tasks are scheduled on the occurrence of some event. Event types are grouped as follows:

1. synchronous
2. asynchronous

Synchronous events represent timed events. These events are predictable in the sense that they are only time dependent. Asynchronous events are those triggered by some external event

or I/O completion. These are time independent events.

Timing is handled by a programmable real time clock in the system. The clock is a register 32 bits long that may be programmed to contain any count between 0 and (2**32)-1. Every clock tick (every 100 micro-seconds) decrements this register by 1. When a 0 to -1 transition occurs an interrupt is generated and control is transferred to the task scheduler.

Each of the EMTs described below enters the task's TCB into a scheduling queue.

5.3.1 Synchronous scheduling

A. AT — A task is started at certain time of day.

B. IN — A task is scheduled at certain increments of time from the instant that it was defined.

C. EVERY — A task is scheduled to start periodically.

D. START — A task is placed into the queue of tasks competing for immediate execution.

5.3.2 Asynchronous scheduling

A. ON — A task is scheduled to start execution upon the occurrence of some event. Such events occur whenever an input-output unit signals the computer that an I/O operation has been completed on a specified unit with a specific return code. Possible return codes might be an end-of-file, an end-of-disk, or a successful I/O completion.

B. WHEN — A task is scheduled to start execution after the A or B interrupt from a device interface card.

Once a synchronous or an asynchronous event occurs the scheduler removes the task's TCB from the scheduling queue and places it into the execution queue.

5.4 Task termination

After a task has been scheduled, it is placed in a scheduling queue. The task waits in this queue until the appropriate event for the task occurs. Once the event occurs, the task is removed from the scheduling queue and placed into the executing queue. The task remains there until all of task code has been executed and all the I/O has been completed. Tasks can be terminated before executing by the CANCEL EMT or before complete execution by the HALT EMT.

The CANCEL EMT call cancels the task. If the task is currently proceeding or has been pre-empted by a higher priority task, during its execution, it will be allowed to complete its current execution. If, however, it is not in the execute queue, it will be cancelled (TCB removed from Scheduling queue) and never be allowed to proceed. Tasks may be rescheduled.

The HALT EMT call can be used to terminate the execution of the currently active task. Tasks waiting in the scheduling queue remain unaffected.

The DESTROY EMT call cancels the task and deallocates the TCB. A DESTROYed task cannot be rescheduled without reDEFINEing the TCB.

All tasks should be DESTROYed when they are no longer needed. Each invocation of a scheduled task requires that a TCB be maintained by the scheduler. There are only 255 of these task control blocks available.

5.5 Locking and unlocking tasks

The LOCK EMT is used to lock the active task into an active state. The task active at time of the call will have its priority raised to 250, effectively making it the highest priority user task and giving it exclusive control. The UNLOCK EMT call will restore the task's original priority. LOCK and UNLOCK can be used to define critical regions, where task interruption is undesirable.

6. OSWIT FILE SYSTEM

6.1 File system overview

OSWIT has mechanisms for creating, destroying, renaming, emptying, truncating, editing and cataloging files on the floppy disk. These mechanisms are provided in the OSWIT system utility programs.

Whenever OSWIT opens a file it creates several internal tables, including a File or Device Usage Block (FDUB - a block which contains information about the file, e.g., logical unit, buffers, etc.). OSWIT defines a file as a seqeunce of logical records placed in 512 byte blocks on the disk. Each block is linked forward and backward with other blocks composing the file. This system allows all files to be accepted as non-contiguous blocks on the disk, thereby avoiding repacking the disk. A file cannot exceed 255 blocks.

Each logical record begins with a single byte containing the true record length between 0 and 255 bytes. A single logical record may overlap two file blocks. Special file control records, such as end of file and checksums, are indicated by using a record length of zero, followed by a code defining the record.

6.2 File naming conventions

File names are limited to 10 characters or less, which consist of any combination of printable, uppercase characters. Lowercase filenames can be created without translation to uppercase, but the OSWIT command intepreter translates the input line to upper case. File names starting with an '*' are designated by OSWIT as public files (see next section). Although the user may define his own public files, they are generally limited to system files.

OSWIT does not support any mechanism for distinguishing data, object and source files. This function is the responsibility of the user. One suggested approach is to append a ".S" for source files, a ".O" for object files, and a ".D" for data files to the file name.

6.3 OSWIT public files

Public files are any filename starting with an '*'. Although the user may define his own public files, this designation is usually reserved for system files and utility programs.

The only functional distinction between public and non-public files is made by *FILESNIFF. Public files are not normally listed in the file directory unless explicitly requested.

The following public files are available:

*ABSLOAD	OSWIT absolute loader
*BLOKEDIT	Disk block editor
*BOOT	Bootstrap for copies of OSWIT other than the one loaded by the hardware bootstrap
*BOOTLOAD	OSWIT software bootstrap
*BITMAP	OSWIT system file indicating the status of blocks on the disk
*CATALOG	OSWIT system file contains the file directory of all files on the disk
*CRZAP	Removes carriage returns from source files originating on MTS
*DISKCOPY	Duplicate disks
*EDIT	Edit disk source files
*FILEFIX	Repairs damaged disk file structures
*FILES11	Manages disk files
*FILESNIFF	Prints file directory
*LINK11	Links relocatable object files into absolute load modules
*LOADCOPY	Used in earlier versions of OSWIT to copy object files from MTS
*OSWIT	Current version of OSWIT
*PATCH	Patches disk absolute load modules
*TEMPFILE1	Temporary scratch file used by utilities
*TEMPFILE2	Temporary scratch file used by utilities
*TEMPFILE3	Temporary scratch file used by utilities
*TIME	Reports the current time of day
*VERIFY	Detects unusable disk blocks

*ABSLOAD, *BITMAP, *BOOTLOAD, *CATALOG, and *OSWIT are system files that must always be present on the system floppy disk. *ABSLOAD and *BOOTLOAD are used to boot the operating

system contained in *OSWIT. *BITMAP is system file that records the current status (used or unused) of all disk blocks. *CATALOG is a system file that contains a directory of all disk files.

In order for the user to utilize disk files, *EDIT, *FILEFIX, *FILES11 and *FILESNIFF should also be present on the floppy disk. These files will allow editing, repairing, file management and cataloging of disk files. In addition, *TEMPFILE1, *TEMPFILE2, AND *TEMPFILE3 are used by some of the utilities as temporary work files. The public utility program files are described in detail in the following chapter.

6.4 File protection

The OSWIT file system does not support any method of protecting files. The hardware, however, provides some degree of protection. All the files on the floppy are protected when the disk is write protected or unprotected when the disk is write enabled. The user should always leave the disk write protected unless he explicitly wants files written.

7. OSWIT UTILITY PROGRAMS

7.1 Utility programs overview

Utility programs provide additional user support not provided in the operating system. These utilities are public files written in CRASH and/or assembly language.

The utility programs are:

*BLOKEDIT	Disk block editor
*BOOT	Bootstrap for copies of OSWIT other than the one loaded by the hardware bootstrap
*CRZAP	Removes carriage returns from source files originating on MTS
*DISKCOPY	Duplicate disks
*EDIT	Edit disk source files
*FILEFIX	Repairs damaged disk file structures
*FILES11	Manages disk files
*FILESNIFF	Prints file directory
*LINK11	Links relocatable object files into absolute load modules
*LOADCOPY	Used in earlier versions of OSWIT to copy object files from MTS
*PATCH	Patches disk absolute load modules
*TIME	Reports the current time of day
*VERIFY	Detects unusable disk blocks

These utilities are described in detail in the following section. The most commonly used utilities are *FILES11, *EDIT, *FILESNIFF and to a lesser extent *TIME and *BOOT. Each of these utilities have a prompt character, different than OSWIT, ODT or the DEBUG package.

7.2 OSWIT utility program descriptions

*BLOKEDIT

Utility Program

Purpose: To allow editing of the data on physical disk blocks.

Logical I/O Units Referenced:

SCARDS - Read user commands
SPRINT - Output information

Description: This is an interactive program to allow display and modification of disk blocks. The program contains a single block buffer that may be displayed, altered, filled with a constant, written to disk, or read from disk. The command summary is as follows:

ALTER <offset> <value> [<value> .... <value>]

This will allow the user to alter consectutive locations in the buffer starting <offset> words from the beginning.

FILL <value>

To fill the buffer with the specified value.

DISPLAY <offset> [<count>]

This will display the specified part of the buffer in octal. Offset is the number of words to offset from the start of the buffer and count is the number of words to be displayed. If count is omitted then 1 word is displayed.

READ <disk addr>

This command will read the specified disk block into the buffer.

WRITE <disk addr>

This will write the buffer into the specified disk block. The third word of the buffer will be altered to contain the correct checksum.

Note - All number are octal.

Example(s):

```
RUN *BLOKEDIT
EXECUTION BEGINS
READ 1000
DISP 10 4
ALTER 12 12 34 -3
WRITE 1004
<CNTL-C>
```

*BOOT

Utility Program

Purpose: To allow copies of the operating system other than the one loaded by the hardware to be used.

Logical I/O Units Referenced:

None.

Description: This program will attempt to load the file given in the PAR field as a new operating system. If there is no PAR field specified, then it will default to booting *OSWIT. If the file specified does not contain a valid operating system then the current core image will be destroyed and you will end up in ODT.

Example(s):
```
RUN *BOOT
RUN *BOOT PAR=*OSWIT24K
```

*DISKCOPY

Utility Program

Purpose: To duplicate a floppy disk.

Logical I/O Units Referenced:

SCARDS - Reads start copy command.
SPRINT - Output messages.

Description: The original disk should be placed in the main drive and the new disk in the auxillary. Also the write protect on the auxillary drive should be off.

When return is hit, the copy operation will start. Due to copying a full track at a time the copy will be completed in about 30 seconds. If any errors occur the copy will be aborted at that point. Entering another return will attempt to recopy the disk.

If control-C is pressed the program will terminate by doing a cold start of OSWIT to restore the disk vectors.

Example(s): RUN *DISKCOPY

*EDIT

Utility Program

Purpose: To edit source files on the floppy disk.

Logical I/O Units Referenced:

SCARDS - Editor command input.
SPRINT - All editor responses.

Description: *EDIT is an interactive program to allow the editing of source files on the floppy disk. The *EDIT program requests the name of the file to be edited. The *EDIT program will take lines from the specified input file, perform the requested operation on the lines and places the edited lines into a *TEMPFILE. Unless the UPDATE command is given, the orginal file remains unchanged. The TOP command is used to re-edit the beginning lines of the file. After each *EDIT command, the lines in the newly edited file are changed accordingly and placed into another *TEMPFILE. The command syntax is as follows:

```
ALTER lpar [<del>string1<del>string2<del>]
DELETE lpar
INSERT lpar
OSWIT
PRINT lpar
REPLACE lpar
SCAN lpar [<del>string1<del>]
TOP
UPDATE
number
```

A description of these commands follows:

ALTER lpar [<del>string1<del>string2<del>]

The ALTER command will change the first occurance of STRING1 to STRING2 in each line for all lines in the given line range. All altered strings are echoed. If an altered string is not entered, the previous altered string will be used. DEL can be any character. If no changes were made the user is prompted for a new command.

DELETE lpar

The DELETE command removes all lines specified by lpar from the file.

INSERT lpar

The INSERT command inserts lines into the file after the line specified by lpar. Fractional line numbers are not allowed. A carriage return will end the insert.

OSWIT

The OSWIT command returns to OSWIT. All changes are stored in *TEMPFILE1 or *TEMPFILE2.

PRINT lpar

The PRINT command is used to print lines in the file specified by lpar.

REPLACE lpar

The REPLACE command will replace the given line number range with the lines entered. The line to be replaced is echoed. The user is then prompted with a question mark for the input line to replace the echoed line. If a carriage return is given on input the replace process stops.

SCAN lpar [<del>string1<del>]

The SCAN command will search the lpar range and print all occurances of STRING1. If STRING1 is omitted, the previous scan string will be used. No checking is made for no lines found. If the search fails, the user is prompted for a new command.

TOP

The TOP command updates the *TEMPFILE and opens it at the top. If the user has done editing and wishes to INSERT at line 0 then he must issue the TOP command.

UPDATE

The UPDATE command updates the file being edited with all changes made. If this command is not entered, the edited file is

stored in either *TEMPFILE1 or *TEMPFILE2 upon return to OSWIT.

number

The 'number' command changes the current line to the current line + number and prints the current line. The number can positive or negitive.

LPAR can be null, contain one line number or contain two line numbers which specify a line range. Line numbers must be non-fractional and positive. Lines are pseudo lines since the files are all sequential. For this reason automatic renumbering takes place after Deletes and Inserts.

Example(s):

```
RUN *EDIT
P 1 5
P 1
A 1 :THIS:
S 1 *END*
10
D 5
UPDATE
```

*FILEFIX

Utility Program

Purpose: To try and repair damaged disk file structures.

Logical I/O Units Referenced:

GUSER - Reads user input to prompt messages.
SERCOM - All printed output from FILEFIX.

Description: This program will trace all files on the disk in the main drive. When an error is found the user is told the file it occured in and the disk address. Then the user will be prompted as to whether the file should be fixed. Fixing consists of chopping the file off at the point of the error and emptying it. If the user reponds yes, the file will be fixed, but if the answer is no, then FILEFIX will attempt to ignore the error and go on. If the error was serious enough, FILEFIX may hang and have to be interrupted. If the reason for answering no was that the data in the file is needed , then the user can try to copy the damaged file to another before running FILEFIX again. This may or may not work depending on the severity of the error.

The user is asked if he wants a full trace map upon program startup. If the answer is yes then every block traced by FILEFIX will be printed. Use of this option by the general user is not recommended since the information is not useful unless manual repair of the file by some other means is to be attempted.

After completing the file trace a check is made to see if there are any files occuping the same disk address. If any are found, the user should destroy both files after FILEFIX is done. Once again, recovery of the data may or may not be possible before the files are destroyed.

The third phase finds and corrects any discrepencies between the bitmap and the actual storage used on the disk. A list of all discrepencies will be printed.

Note - FILEFIX does not currently correct any erroneous information in the catalog. So if a

FILESNIFF bitmap and catalog block count disagreed before the fix they will still disagree.

Example(s): RUN *FILEFIX

*FILESNIFF

Utility Program

Purpose: To print information about the files on the disk in the main drive.

Logical I/O Units Referenced:

SPRINT - The file information
SERCOM - Fatal open error messages(should not occur)

Parameters: The par field is used to specify what files are to be sniffed.

Description: If no par field is specified then all files that do not begin with an asterisk will be sniffed.

When "PAR=*" is specified all files on the disk will be sniffed. Also the number of blocks allocated to all files listed in the catalog, number of blocks listed in the bitmap, and the number of free blocks remaining will be printed.

Note - for "PAR=*" the number of blocks listed in the catalog should equal the number of blocks in the bitmap. If they do not then the file structure on the disk is defective.

If "PAR=filename" is specified then only that particular file is sniffed. If the file doesn't exist then an appropriate message is printed out.

When "PAR=FI?" is specified, all files beginning with the characters "FI" are sniffed. The "?" may appear only as the last character of the file name fragment. If"PAR=?" is specified, the resultant action is the same as "PAR=*".

When "PAR=FULL" is specified, all non-asterisk files are sniffed as when no "PAR=" field is specified. However, extended file information is printed out - file size, truncated size, extended size, and start address. When "FULL" is appended to any parameter field, this extended information is printed out.

Example(s):

```
RUN *FILESNIFF PAR=*

RUN *FILESNIFF PAR=OS?FULL

= (FILE NAME) (FILE SIZE) (TRUNC SIZE) (EXT SIZE)

= OS.V4.0          60             45          60

= OS.V4.1          64             64          60

= OS.V4.2          66             46          66
=150 BLOCKS LISTED IN *CATALOG
=250 BLOCKS LISTED IN *BITMAP
=232 FREE BLOCKS

RUN *FILESNIFF SPRINT=FILEINFO PAR=PROBLEM1

RUN *FILESNIFF PAR=FULL

= (FILE NAME) (FILE SIZE) (TRUNC SIZE) (EXT SIZE)

= X                10             3           10

= SWFLIP            1             1            1

= TRAININT          1             1            1

= OS               60            45           60

= V3.1             64            64           60

= V2.6             45            45           70
=181 BLOCKS LISTED IN *CATALOG
=265 BLOCKS LISTED IN *BITMAP
=247 FREE BLOCKS
```

*FILES11

Utility Program

Purpose: To allow the user to create, destroy, empty, truncate, and rename files.

Description: *FILES11 is an interactive program to allow file manipulation by the user. The command syntax is as follows:

```
CREATE filename [size]
DESTROY filename [!,OK,O.K.]
EMPTY filename [!,OK,O.K.]
OSWIT
RENAME filename1 filename2 [!,OK,O.K.]
TRUNCATE filename
```

A description of these commands follows:

CREATE filename [size]

The CREATE command is used to create a file on the floppy disk. If size is not specified, then it is created with a size of 1 block (512 bytes long). Otherwise it is created with the number of blocks specified. The initial size of the file is also the amount by which the file will be extended when necessary. The size of a file cannot exceed 255 blocks. The filename is limited to 10 characters in length. If the specified filename already exists then an error message will be printed.

DESTROY filename [!,OK,O.K.]

The DESTROY command will remove a filename entry from the catalog and release its blocks to the free block pool. If confirmation is not given on the destroy command a prompt will be issued. Any responses other than expected will cause the command not to be executed. An error will occur if the filename specified does not exist. If the optional parameters are omitted, then confirmation is required before the file is destroyed.

EMPTY filename [!,OK,O.K.]

The EMPTY command will empty a file by writing an end of file mark as the first thing in the file. The size of the file does not change when it is emptied, but the truncate size is set to 1. If the specified filename does not exist then an error message will be printed. Confirmation is required as in the destroy command.

OSWIT

The OSWIT command will return control to OSWIT without unloading the program. If a end-of-File is typed (cntl-c) then the program will be unloaded and control passed to OSWIT.

RENAME filename1 filename2 [!,OK,O.K.]

The RENAME command will change the name of a file from filename1 to filename2. An error occurs if filename1 does not exist or if filename2 already exists. Confirmation is required as in the destroy command.

TRUNCATE filename

The TRUNCATE command is used to free unused blocks at the end of a file. The freed blocks are returned to the pool of available blocks. If the specified filename does not exist then a error message will be printed. An empty file has a size of 1.

Example(s): RUN *FILES11

```
CREATE FILE1
CREATE OXSNARD 50
DESTROY OXSNARD OK
TRUNCATE FILE1
EMPTY FILE1 O.K.
RENAME FILE1 PTAVV !
```

*LINK11

Utility Program

Purpose: To link relocatable object files into an absolute load module.

Logical I/O units Referenced:

GUSER - Commands for the link-editor are read through this unit.
SERCOM - Errors and other linkage information are output here.

Description: The link editor is designed to be an interactive program, although it can also read it's commands from devices other than the console by reassigning GUSER. This program requires a fairly large amount of memory to operate (at least a 24K system), so as much memory should be made available as possible (eg. SET DEBUG=OFF). A stack size of 1024 bytes is sufficient.

The basic commands accepted by the link editor are as follows:

```
CLEAR
LINK <FILENAME>
MAP [<FILE OR DEVICE>]
SET <SYMBOL> <VALUE>
SYMBOL <SYMBOL NAME>
STOP
WRITE <FILE OR DEVICE>
?
*
```

A description of these commands follows:

CLEAR

To clear out any previously linked programs from the link editors symbol table. This command is done automatically when the program is started, it is needed only to generate multiple load files or to erase bad input from the symbol table.

LINK <FILENAME>

This command will cause the specified file or device to be read and all of it's symbol and relocation information to be stored. the filename is also stored for use when a write command is issued.

MAP [<FILE OR DEVICE>]

To produce an octal load map of all the symbols currently in the symbol table. If the optional file or device name is given then the map will be written on that file or device.

SET <SYMBOL> <VALUE>

This command is used to set the current load address and entry point. If the symbol "@" is specified the next module to be linked will be linked starting from the specifed address. As each module is read the "@" pointer is automatically advanced. The default value of this symbol is octal 400. It is set to this value when the program is started or when a clear command is issued.

If the symbol "#" is specified the entry point will be set to the specified value. The entry point is set to zero when the program is started and when a clear command is issued. This value will be set to the start of the first csect if it has not been changed by an entry on an END card or by the set command. The final value will be that given on the last end card read or last entry command given. If neither of these modified the at point then it will point to the first csect linked.

STOP

To terminate execution of the program. Reading an endfile from GUSER will have the same effect.

SYMBOL <SYMBOL NAME>

This command will cause the value of the

specified symbol to be printed. If the the symbol is currently undefined then an appropriate message is printed.

WRITE <FILE OR DEVICE>

When the write command is issued a check will be made to see if all symbols have been resolved. Any unresolved symbols will be printed and the write will be aborted.

A second pass is made through all the files to extract all the text information. As this information is read the load module is generated and written on the specified file or device.

?

This command will cause a list of all unresolved symbols to be printed.

*

Any text folowing an asterisk is regarded as a comment and will be ignored.

Note - there must be at least one blank immediately after the asterisk.

Example(s):

```
RUN *LINK11
LINK SERVO.OBJ
WRITE SERVO.LOAD
MAP SERVO.MAP
STOP

RUN *LINK11
SET @ 400
SET # 1000
LINK FILE1
?
* THE UNRESOLVED SYMBOLS WOULD BE PRINTED HERE.
LINK FILE2
SET @ 10000
LINK FILE3
WRITE FILE4
CLEAR
LINK ZZZ
WRITE YYY
STOP
```

*LOADCOPY

Utility Program

Purpose: To read a load file byte by byte and convert it to load records.

Logical I/O Units Referenced:

SCARDS - The load file to be copied.
SPUNCH - Where to write the load records.
SERCOM - Completion and error messages.

Description: This program will input a byte oriented load file and output a record oriented load file. The paper tape reader(*READER*) and MTS(*MTS* OSWIT version 2.6) are byte oriented devices. It is illegal to just issue a copy on these devices. The input is checked for the following errors:

1 - Checksum errors.
2 - Premature end of file on scards
3 - Missing or invalid start record.

Example(s): RUN *LOADCOPY SCARDS=*MTS* SPUNCH=MYLOADFILE

RUN *LOADCOPY SCARDS=*READER* SPUNCH=ATAPELOAD

RUN *LOADCOPY SCARDS=NEWOS SPUNCH=*OSWIT SERCOM=*DUMMY*

*PATCH

Utility Program

Purpose: To make patches to floppy files containg absolute load modules.

Logical I/O Units Referenced:

GUSER - To read patch statements from the user.

Description: When started the program will be in file request mode. The user should enter the name of the file to be patched. If a legal name is entered then patch mode will be entered. Otherwise a new filename will be requested. If a control-c is entered the program will terminate execution.

There are two legal commands in patch mode, ALTER & CSECT. These commands operate identically to those in the OSWIT debugger. All ALTER commands are converted to load records and stored in the file being patched so the next time the file is loaded the patches will be in place. If a control-C is entered while in patch mode, control will revert to file request mode.

Example(s):

```
RUN *PATCH
EXECUTION BEGINS
ENTER NAME OF FILE TO BE PATCHED.
?SUPERTRAIN
) CSECT 400
) ALTER +20 137 +1254
) AR 20 100
) [CNTL-C]
ENTER NAME OF FILE TO BE PATCHED.
?[CNTL-C]
EXECUTION TERMINATED
```

*TIME

Utility Program

Purpose: To print the current time of day.

Logical I/O Units Referenced:

SPRINT - all program output

Description: The contents of the time of day clock will be converted to a text string and written out.

Example(s):

```
RUN *TIME
EXECUTION BEGINS
TIME=21:34:16.3945
EXECUTION TERMINATED
```

*VERIFY

Utility Program

Purpose: To detect unusable blocks on a floppy disk.

Logical I/O Units Referenced:

SPRINT - All bad block information output.

Description: This routine will scan the disk in the main drive for bad disk blocks. It will first read the block to be tested into a buffer. Then a test pattern will be written out and verified. Next the complement of the test pattern wil be written and verified. Finally the original block contents will be written out and verified.

The message "read error" will be printed if the block contained a bad checksum. The message "write error" will be printed every time a write-verify operation fails.

This program takes about 9 minutes to run because of the large number of I/O operations. It is not recommended that this program be interruped since a test pattern could get left in an unfortunate spot on the disk.

Example(s): RUN *VERIFY

Appendix A: OSWIT ERROR MESSAGES

Illegal memory reference (bus time out)
Illegal instruction
Floating point exception
Bad EMT number
Buffer system impure
There is nothing to restart
Command unknown
End of disk on write
Buffer overflow
Stack size too big
Error in set command
Incorrect format for filename
File doesn't exist
MTS is not connected
Missing or bad parameter or filename
Checksum error during loading
Missing or invalid start address
There is no program loaded
Input line len >255
Illegal unit
Illegal file assignment string format
Tasker - no buffers
Tasker - undefined ID
Tasker - time CO
Tasker - stack error
***Task rescheduling has occurred
Illegal device for EMT WHEN
No buffer for I/O queueing
Unit NN: bad I/Otype
Byte bof has run out of buffer
RC not 2 when I/O queued
Disk read error
Write verify error
Bad file structure
Open getbuf error
Disk removed while no I/O
Grow getbuf error
SRCHCAT catalog file error
Squeeze gelfat error
Squeeze disk read error
Finfo getbuf error
Finfo disk read error
Finfo disk write protect
Finfo write error - will retry

Appendix B: OSWIT SYSTEM DIRECTIVES

7.3 System directive overview

All LSI-11 operation codes from 1040008 to 1043778 are EMT instructions. These instructions represent supervisor "calls" which transmit information to OSWIT to perform predefined system functions. When an EMT is encountered, the processor traps to the EMT trap vector located at 308, loads the EMT handler routine address from 308 and the new processor status word from 328 and executes the EMT routine handler. This handling routine in OSWIT transfers control to specified routine to perform the pre-defined system function. Control is subsequently returned to instruction after the EMT instruction.

For a complete description of the EMT instruction and the LSI-11 interrupt facility see DEC's Microcompuer Manual, 1976.

The next section describes the EMT routines incorportated into OSWIT.

7.4 EMT Descriptions

AT

EMT Routine

Purpose: To schedule a task at a particular time of day.

Calling Paramters:

```
  (SP)+4
: lo 16 bits of time of day

  (SP)+2
: hi 16 bits of time of day

  (SP)
: task ID
```

Remarks: This routine will schedule a task to be started at the time of day specified. The time is the number of hundreds of micro-seconds since midnight. It must be a positive 32 bit integer less than 864,000,000(24*3600*10000).

The EMT number of this routine is 42.

Example(s):

```
*
*     6 PM = 18*3600*10000 = 9887*(2**16)+45568
*
          MOV   =45568,-(SP)      PUSH TLO
          MOV   =9887,-(SP)       PUSH THI
          MOV   ID,-(SP)          PUSH TASK ID

          EMT   AT                SCHEDULE THE TASK
```

Error(s): An error will occur if an illegal ID is passed to the tasker.

BIN2O

EMT Routine

Purpose: To convert one binary word into octal ASCII characters.

Calling parameters:

SP+4
: word to be translated

SP+2
: size of buffer area

SP
: address of buffer area

Return parameters:

The converted word is returned in the buffer and the stack is clear.

Remarks: The word is translated into octal ASCII characters and placed right justified into the field. If the field is too small it will be filled with asterisks. If the field is too large it will be padded on the left with blanks

The EMT number for this routine is 1.

Example(s):

```
        MOV   VALUE,-(SP)
                        PUSH VALUE
        MOV   =BUFFER,-(SP)
                        PUSH BUFFER ADDRESS
        MOV   =2,-(SP)  PUSH SIZE
        EMT   BIN2O     GO DO IT

VALUE   DC    O'177777'
BUFFER  DS    2C        BUFFER
```

The result of this run will place ** in buffer because the field is too small.

Error(s): The field will be filled with asterisks on field overflow.

CANCEL

EMT Routine

Purpose: To cancel any further executions of a particular definition of a task.

Calling Paramters:

(SP) : task ID

Remarks: This routine will cancel any further executions of the specified ID. If the task is currently executing it will be allowed to finish. The TCB will be removed from all scheduling queues.

The EMT number of this routine is 40.

Example(s):

```
          MOV   ID,-
(SP)            PUSH THE TASK ID

          EMT   CANCEL              CANCEL THE TASK
```

Error(s): An error will occur if a illegal task ID is passed.

CLOSE

EMT Routine

Purpose: To disconnect an internal unit number from a file or device, and reconnect the unit to the default device (*MSOURCE* or *MSINK*).

Calling parameters:

```
SP
: the unit number
```

Return parameters:

None

Remarks: A single unit specified must be between 0 and 30.

All units may be closed by specifying a negative unit number.

The EMT number for this routine is 15.

Example(s):

```
MOV   =10,-(SP) CLOSE LOGICAL UNIT
EMT   CLOSE
BCS   BADUNIT   IF ERROR
```

Error(s): If an illegal unit number occurs the C BIT is set to "1".

DEFINE

EMT Routine

Purpose: To create a task control block (TCB) and enter it in the DEFINED queue.

Calling parameters:

(SP)+6 : processor status to start task with
(SP)+4 : priority of task (0-255)
(SP)+2 : starting address of task being defined
(SP) : address of word to store task ID in

Return paramters:

The task ID is returned in the specified location.

Remarks: This routine makes a task occurence known to the operating system. A TCB is allocated and entered into the defined queue. The address of the TCB will be returned in the location specified for the ID. The ID of the main task will be in register 0 when it is started by a RUN or a START command (Note - the address of the PAR string is in R1).

The starting address of the task is where a task will be entered when it becomes active. Multiple definitions of a single task can be accomplished by making more than 1 call to DEFINE with the same starting address since a task is identified by its ID, not by its starting address. This is useful if it is desired to initiate a task on more than one occurence of an event type. Examples might be to have a task scheduled every 3 seconds and every 5 seconds, or to schedule a task on multiple WHEN conditions. Event types are grouped as follows:

1) timed events
2) ON and WHEN events

A single task definition may be queued on only

one of each event type. To go higher you need multiple definitions. However to use multiple TCB's the task must be reentrant.

A single task definition may be scheduled for more than one execution. This can come about by a task scheduling itself, a burst of events, or a scheduling request coming in while the task is blocked by a higher priority task. The maximun number of times a single TCB can be scheduled at one time is 255. Any further requests are ignored until the count drops below 255.

The priority determines the order that tasks are executed once scheduled. Scheduled tasks will be activated in order of decreasing priority. Within a priority level they are started in the same order they were scheduled. Scheduling policy is such that no lower priority task can take control from a higher or same priority task, even if the higher or same priority task is blocked by I/O or some other process. It is recommended that the user tasks restrict themselves to priorities 5 through 250. The other priorities are reserved for system use. The main task will be started with a priority of 10.

The processor status allows tasks to be started with interrupts ON or OFF. Except for special cases tasks should run with interrupts ON.

The EMT number for this routine is 33.

Example(s):

```
          CLR   -
(SP)              PUSH PROCESSOR STATUS(ION)
          MOV   =25,-(SP)          PUSH PRIORITY
          MOV   =TASK,-
(SP)         PUSH ADDRESS OF TASK
          MOV   =ID,-
(SP)           PUSH ADDRESS OF ID
          EMT   DEFINE             DEFINE THE TASK

TASK      EQU   *                  ENTRY POINT OF T

ID        DS    H                  STORAGE FOR TASK
```

DESTROY

EMT Routine

Purpose: To cancel any further executions of a task definition and to destroy the TCB.

Calling Paramters:

(SP) : task ID

Remarks: This routine will cancel any further executions of the specified ID. If the task is currently executing it will be allowed to finish. As soon as it finishes the TCB will be destroyed(the ID will no longer be valid).

The EMT number of this routine is 41.

Example(s):

```
(SP)        MOV   ID,-
                  PUSH THE TASK ID

            EMT   DESTROY          DESTROY THE TASK
```

Error(s): An error will occur if an illegal ID is passed.

DSKIO

EMT Routine

Purpose: DSKIO performs physical block transfers with the floppy disk unit. This is not the normal way of doing I/O to the disk (see instead OPEN, READ, WRITE, WAIT, etc.)

DSKIO handles queing of all I/O for a given disk unit, on strictly a first in, first served basis. It will perform three commands: READ, WRITE, and WRITE-VERIFY. DSKIO does all handling of the block checksum. It generates it for a WRITE or a WRITE-VERIFY, and checks it on a READ. A WRITE-VERIFY operation is accomplished by a WRITE followed by a READ, and then a word by word comparison.

Calling parameters:

SP
: address of DSKBUF (see figure 1)

R1
: points to PHYTAB entry for disk

Return parameters:

None (from the EMT)

Remarks: DSKIO queues the request and does an immediate return to the calling task. Upon the interrupt from the disk after a READ, WRITE or the READ portion of a WRITE-VERIFY operation: DSKIO will check the checksum and verify as necessary, and then do a JMP to the user's program at the address specified by DSKENT (see figure 1). On the stack will be:

SP : ADDRESS OF dskbuf

SP+2 : ADDRESS OF THE phytab FOR THE DISK

SP+4 : PC of interrupted task

SP+6 : PS of interrupted task

The current PSW will indicate errors as follows:

C-BIT set :
the checksum is incorrect on a READ or the block does not verify on a WRITE-VERIFY.

V-BIT set :
unable to get required buffer to perform read back and verification on a WRITE-VERIFY.

data format:

the parameters for DSKIO ar organized as in figure 1, usually in the form of a DSECT.

```
  DSKBUF
DSECT

  DSKLINK
DS    A          LINK TO NEXT I/O REQUEST

  DSKCMND
DS    F          COMMAND TO PERFORM

  DSKCNT
DS    F          # OF WORDS TO TRANSFER

  DSKENT
DS    A          ADDRESS TO GO TO WHEN DONE

  DSKBLOCK
EQU   *          DISK I/O STARTS HERE

  DSKLINKB
DS    F          BACKWARD LINK (DISK ADDR)

  DSKLINKF
DS    F          FORWARD LINK (DISK ADDR)

  DSKCHKSM
DS    F          CHECKSUM

  DSKDATA
DS    253F       ACTUAL DATA GOES HERE
```

figure 1

DSKLINK is strictly for DSKIO's internal use in queing up requests.

DSKCMND is the command DSKIO is to perform, which is specified as follows:

X XYY YYZ ZZZ ZZZ ZZ0

where XX is:

00 READ

01 WRITE

10 READ-(NO CHECKSUM CHECKING)

11 WRITE-VERIFY

where YYYY is reserved for future expansion ?????

where Z ZZZ ZZZ ZZ0 is the block number to transfer

DSKCNT is the number of words to transfer: 3DSKCNT256. specifing DSKCNT<256 is equivalent to padding the last 256-DSKCNT words with 0's (on a write only). The DSKCNT specified for a READ should be the DSKCNT specified when that block was written, otherwise a checksum error will (should) occur.

DSKENT is the address jumped to by the interrupt handler when this I/O request is complete (see operation description).

DSKBLOCK The next DSKCNT words are what actually get transfered to/from the disk.

DSKLINKB is the pointer to the previous disk block in a logical file structure.

DSKLINKF is the pointer to the next disk block in a logical file structure. DSKIO will, when there is no quequed I/O, perform a SEEK on the block specified by DSKLINKF (if non-null) of the last block transfered. If the DSKLINKB and DSKLINKF are not being used, they should be set to null links (f"-1").

DSKCHKSM is the checksum returned from a READ or space for the checksum generated on a WRITE.

DSKDATA is DSKCNT-3 words of whatever you like.

The EMT number for this routine is 31.

Example(s):

```
        MOV   @=DISKVEC,R0      GET ADDR OF DISK
```

```
                TST     (R0)+               BUMP PTR TO POIN

                MOV     @R0,R1              FETCH PHYTAB ADD
                MOV     =MYBUF,-
(SP)              PUSH DSKIO BUFFER ADDR
                CLR     FLAG                INIT DONE FLAG
                EMT     DSKIO               START THE I/O

                TST     FLAG                CHECK IF I/O DON
                BEQ     *-4                 LOOP IF NOT


INT             CMP     (SP)+,(SP)+         THROW AWAY PHYAD
                INC     FLAG                SET DONE FLAG
                RTI                         RETURN

FLAG            DS      F

MYBUF           DS      F                   LINK SPACE FOR D
                DC      O'1776'             READ BLOCK 1776

                DC      F"256"              256 WORDS TO TRA

                DC      A"INT"              INTERRUPT HANDLE

                DS      256F                THE ACTUAL DATA
```

D2BIN

EMT Routine

Purpose: To convert an ASCII character string of decimal digits to its equivalent two's complement value.

Calling parameters:

SP+2
: field length to scan

SP
: address of word to convert

Return parameters:

If the C BIT is clear then:

SP
: the binary value of the number converted

SP+2
: the address of the character which caused conversion to stop

otherwise the stack is clear.

Remarks: The first characters may be blanks, a + sign, or a minus sign. Thereafter all characters must be decimal ASCII characters until the number terminates, or the field length is exhausted. The range on the numbers to convert is 32767 to -32767.. If no error has occurred (C BIT clear) the address of the character which caused conversion to stop, or the address of the byte following the field length specified, is placed on the stack. The converted number is placed on top of the stack. If the number is out of range (C BIT set) the stack is clear (no number is returned). If a nondigit was encountered before the end of the string, the converted number is returned as above, but the V BIT will be set and the C BIT will be clear. Then if desired, the user may test for this condition.

The EMT number for this routine is 2.

Example(s):
```
MOV   =3,-(SP)  PUSH FIELD LENGTH
MOV   =BUFFER,-(SP)
                PUSH BUFFER ADDRESS
EMT   D2BIN     GO DO IT
BCC   OK        NO ERRORS
```

```
            JMP    ERRORHAN  TO ERROR HANDLER
OK          MOV    (SP)+,VALUE
                              GET VALUE
            MOV    (SP)+,NEXTCHAR
                              GET BREAK ADDRESS
            ..

BUFFER      DC     C' 167'
VALUE       DS     F
NEXTCHAR    DS     F
```

This routine will return a binary 16 on top of the stack, followed by the address of the "7".

Error(s): A number outside of range (32767 to -32767) is shown by setting the C BIT to "1".

ERROR

EMT Routine

Purpose: To output a standardized error message followed by a call to OSWIT for user interaction.

Calling parameters:

SP+2
: logical unit number

SP
: the buffer address

Return parameters:

None

Remarks: The logical unit specified must be between 0 and 30.

This routine will return control to OSWIT, however control may be returned to the user program via a RESTART command.

The first byte of the buffer must specify the buffer length.

The EMT number for this routine is 19.

Example(s):

```
        MOV    =SERCOM,-(SP)
                        UNIT NUMBER
        MOV    =ERRORMSG,-(SP)
                        ADDRESS OF MESSAGE
        EMT    ERROR
        ..
ERRORMSG DC    H"24",C'THIS IS AN ERROR MESSAGE'
```

this call will print out:

***ERROR, THIS IS AN ERROR MESSAGE

control will be passed to OSWIT

Error(s): None

EVERY

EMT Routine

Purpose: To reschedule a task repeatedly at a fixed time interval.

Calling Paramters:

```
(SP)+4      : lo 16 bits of time interval
(SP)+2      : hi 16 bits of time interval
(SP)        : task ID
```

Remarks: This routine will schedule a task every N ticks of the real time clock (a clock tick occurs every 100uS). n is a non-zero positive 32 bit integer.

The EMT number of this routine is 35.

Example(s):

```
          MOV   =10000,-
(SP)        PUSH LO TIME(1 SECOND)
          MOV   =0,-(SP)          PUSH HI TIME
          MOV   ID,-(SP)          PUSH TASK ID

          EMT   EVERY             PUT TASK IN TIME
```

Error(s): An error will occur if an illegal ID is passed to the tasker.

EXIT

EMT Routine

Purpose: To return all resources to the operating system.

Calling parameters:

None

Remarks: Program execution is terminated and all memory space is returned to the system. All logical I/O units are reset to *MSOURCE* or *MSINK*. Control is not returned to the user but rather to OSWIT.

The EMT number for this routine is 23.

Example(s): EMT EXIT TO SYSTEM

Error(s): None

GETBUF

EMT Routine

Purpose: To have the operating system allocate a temporary buffer for the user.

Calling parameters:

SP
: number of bytes desired

Return parameters:

SP
: address of the first word in the buffer if the C BIT is set to "0"; otherwise the stack is clear.

Remarks: The calling parameter is the number of bytes desired. All allocations begin on a word boundary and are an integral number of words. The address returned is the word address of the first word in the buffer.

the EMT number for this routine is 5.

Example(s):
```
MOV   =160,-(SP)
                  ASK FOR 160 BYTES
EMT   GETBUF      GO GET IT
BCS   NOSPACE     NO SPACE LEFT
MOV   (SP)+,STRADR
                  GET STARTING ADDRESS
```

This example will request from the operating system 160 contigious bytes (80 words) of storage and if this amount of storage exists will place the starting address in STRADR.

Error(s): The C BIT is set to "1" if there is not enough space to accommodate the request.

GETPAR

EMT Routine

Purpose: To build a stack of ADCONS which point to parameters and modifier names in a string given it for parsing.

Calling parameters:

SP+2
: string starting address (byte containing length)

SP
: address of where search is to begin

Return parameters:

SP
: address of break character

SP+2
: address of first non-break character

SP+4
: number of modifier names

SP+6
: addresses of each modifier name are on stack

Remarks: GETPAR expects the first byte of string to contain the length of string followed by bytes of data to be parsed.

Returned on the top of the stack is the address of the next byte to be parsed followed by the address of the first parameter seen, which is followed by the number of modifier names followed by an address for each modifier name.

****** break characters are: " " and ","

****** modifier characters are: "@" and "="

The EMT number for this routine is 9.

Example(s):
```
MOV   =STRING,-(SP)
                  ADDRESS OF THE STRING
MOV   =STRING+1,-(SP)
                  WHERE TO START SCANNING
EMT   GETPAR
BCS   NOPARAM
..
```

```
STRING   DC    H"10"
         DC    C' FILE@BIN '
```

In this example(s):

FILE is the parameter

@BIN is a modifier

@ is the modifier character

BIN is the modifier name

This example(s): will return the address of the blank following the "N" on the top of the stack followed by the address of the "F" on the stack followed by a 1 to indicate one modifier name, followed by the address of the "B".

Error(s): If nothing is left in the string the C BIT will be set to "1" and the stack will be clear.

HALT

EMT Routine

Purpose: To terminate the current execution of a task.

Calling Paramters:

None

Remarks: This routine will terminate execution of the task that was active at the time of the call. If the active task had multiple executions scheduled then it will be restarted and the execution count decremented (except if it had been cancelled). If the task was scheduled for destruction then its TCB will be destroyed at this time. when there are no tasks left able to execute the tasker will return control to OSWIT, otherwise it will wait.

The EMT number of this routine is 39.

Note - this routine should be used to terminate all tasks. If EXIT is used instead then all tasks in the system will be destroyed and control returned to OSWIT.

Example(s):

```
EMT   HALT                TERMINATE EXECUT
```

Error(s): An error will occur if the stack has had more items popped off than were on it when the task was started.

IN

EMT Routine

Purpose: To schedule a task for execution in a certain amount of time.

Calling Paramters:

```
(SP)+4      : lo 16 bits of time interval
(SP)+2      : hi 16 bits of time interval
(SP)        : task ID
```

Remarks: This routine will schedule a task in N ticks of the real time clock (a clock tick occurs every 100uS). n is a non-zero positive 32 bit integer.

The EMT number of this routine is 34.

Example(s):

```
          MOV   =10000,-
(SP)        PUSH LO TIME(1 SECOND)
          MOV   =0,-(SP)          PUSH HI TIME
          MOV   ID,-(SP)          PUSH TASK ID

          EMT   IN                PUT TASK IN TIME
```

Error(s): An error will occur if an illegal ID is passed to the tasker.

LISTEN

EMT Routine

Purpose: OSWIT command language handler.

Calling parameters:

None

Return parameters:

None

Remarks: A call to this routine will invoke OSWIT's command language handler. LISTEN will return to the user's program if he enters a RESTART command.

The EMT number for this routine is 18.

Example(s): EMT LISTEN

Error(s): None

LOAD

EMT Routine

Purpose: To load a file into main memory.

Calling parameters:

```
SP
: filename
```

Return parameters:

```
SP
: the load address if the C BIT is set to "0";
     otherwise an error code
```

Remarks: The filename string must begin with one byte giving the length of the filename. The contents of the named file are loaded into main memory. however, execution is not begun.

The EMT number for this routine is 7.

Example(s):

```
        MOV   =STRING,-(SP)
                      FILE ZIPPY
        EMT   LOAD    IS TO BE LOADED
        BCC   OK      NO ERRORS
        ADD   PC,@SP  BRANCH TO
        MOV   @(SP)+,PC ...ERROR ROUTINE:
        DC    A"NOFILE"   NON-EXISTANT FILE
        DC    A"BADUNIT"
                          ILLEGAL UNIT NUMBER
        DC    A"BADFMT"   ILLEGAL FORMAT
        DC    A"BADSUM"   CHECKSUM ERROR
OK      MOV   (SP)+,LOADADR
                      SAVE ADDRESS
        ..
STRING  DC    H"8",C'TESTPROG'
LOADADR DS    F
```

Error(s): If an error occurs the C BIT is set to "1" and one of the following error codes placed on the stack:

```
2
: non-existant file

4
: illegal unit number

6
: bad filename format
```

```
  8
: checksum error
```

LOCK

EMT Routine

Purpose: To lock the active task into the active state.

Calling Paramters:

None

Remarks: The task that was active at the time of the call will have it's priority raised to 250. This effectivly makes it the highest priority user task giving it exclusive control of the CPU.

The EMT number for this routine is 44.

Example(s):

```
EMT    LOCK                    GET EXCLUSIVE CPU
```

Error(s): None

OPEN

EMT Routine

Purpose: To provide a connection between an internal unit number and a file or device.

Calling parameters:

SP
: the address of the assignment string

Return parameters:

(SP) :return code if VBIT is set, otherwise stack is clear.

Remarks: The length of the string is contained in the first byte of the string.

The assignment string must be of the form:

unit#=pseudodevice

Where unit# can be 0 - 30 or one of the following equivalent strings.

Device equivalences are provides to be compatible with mts as follows:

SCARDS = 26

GUSER = 27

SPRINT = 28

SPUNCH = 29

SERCOM = 30

If the unit was already opened, it is closed, and then reopened.

The EMT number for this routine is 14.

Example(s):

```
MOV   =STRING,-(SP)
                  OPEN SCARDS
EMT   OPEN        ... WITH *MTS*
BVC   OPENOK      BRANCH IF NO ERROR
ADD   PC,@SP      VECTOR TO
```

```
          MOV   @(SP)+,PC PROPER HANDLER
          DC    A"PDNEXIST"
          DC    A"BADSTR"
          ..
STRING    DC    H"12",C'SCARDS=*MTS*'
```

Return codes:

4 : File or pseudo-device non-existant

6 : Illegal string format

8 : Bad unit number

O2BIN

EMT Routine

Purpose: To convert an ASCII character string of octal digits to its equivalent two's complement value.

Calling parameters:

SP+2
: field length to scan

SP
: address of word to convert

Return parameters:

if the C BIT is clear then:

SP
: the binary value of the number converted

SP+2
: the address of the character which caused conversion to stop;

Otherwise the stack is clear.

Remarks: The first character may be blank, a plus, or a minus sign. Thereafter all characters must be octal ASCII characters until the number terminates, or the field length is exhausted. The range on the number to convert is 177777 to 0. If no error has occurred (C BIT is clear) the address of the character which caused the conversion to stop, or the address of the byte following the field length specified is placed on the stack. The converted number is placed on top of the stack. If the number is out of range (C BIT set) the stack is clear (no number is returned). If a nondigit was encountered before the end of the string, the converted number is return as above, but the V BIT will be set and the C BIT will be clear. Then if desired, the user may test for this condition.

The EMT number for this routine is 3.

Example(s):

```
MOV   =4,-(SP)   PUSH FIELD LENGTH
MOV   =BUFFER,-(SP)
                 PUSH BUFFER ADDRESS
EMT   O2BIN      GO DO IT
BCC   OK         NO ERRORS
```

```
            JMP   ERRORHAN  TO ERROR HANDLER
OK          MOV   (SP)+,VALUE
                            GET VALUE
            MOV   (SP)+,NEXTCHAR
                            GET ADDRESS OF BREAK
            ..

BUFFER      DC    C' 167'
VALUE       DS    F
NEXTCHAR    DS    F
```

This routine will return a octal 167 on top of the stack.

Error(s): Number outside of range (177777 to 0) shown by C BIT being set to "1".

PARSE

EMT Routine

Purpose: This is a general table driven parser routine. It is capable of parsing input character by character from the keyboard or a line at a time from a core buffer.

Calling parameters:

```
  SP+4
: string lengt

  SP+2
: string address

  SP
: table address
```

Return parameters:

```
  SP
:  2*  the number of the string that matched if the
                    C BIT is set to "0";
                    otherwise the stack is clear
```

Remarks: The "PARSE" EMT requires a table of strings against which it will try to match the input string. Each string in the table must be preceeded by one byte which contains the length of that string. A null string (length=0) indicates the end of the table.

PARSE will compare the input string with the entries in the table in the order in which entries appear in the table. If a match is not unique (e.g. RE will match RESTART or RESTORE), PARSE will return the first match it finds in the table. If no match is found, the C BIT is set to "1". If one is found, it is set to "0", and 2* the string number in the table will be returned on top of the stack.

The EMT number for this routine is 8.

Example(s):

```
MOV   =LENGTH,-(SP)
                  PUSH STRING LENGTH
MOV   =STRING,-(SP)
                  PUSH STRING ADDRESS
MOV   =CMDTABLE,-(SP)
                  PUSH TABLE ADDRESS
EMT   PARSE       LOOK FOR MATCH
```

```
            BCS   BADCMD     BRANCH IF NO MATCH
            ADD   PC,@SP     BRANCH TO
            MOV   @(SP)+,PC SERVICE ROUTINE FOR
            DC    A"CMD1"    ...CMD1
            DC    A"CMD2"       CMD2
            DC    A"CMD3"       CMD3
            ..
CMDTABLE    DC    A1"XRESTART-CMDTABLE-1"
            DC    C'COPY'
XRESTART    DC    A1"XRESTORE-XRESTART-1"
            DC    C'RESTART'
XRESTORE    DC    A1"XEND-XRESTORE-1"
            DC    C'RESTORE'
XEND        DC    H"0"
            ..
STRING      DC    C'RES'
LENGTH      EQU   *-STRING
```

The preceeding example will return a 4 on the top of the stack to indicate a match of the second string.

Error(s): If there was no match the C BIT is set to a "1".

READ

EMT Routine

Purpose: To read a logical record from a device.

Calling parameters:

SP+2
: logical unit number

SP
: the buffer address

Return parameters:

None

Remarks: The logical unit specified must be between 0 and 30.

Although the routine read returns immediately, the transmission is not necessarily complete.

Upon return the first byte of the buffer will contain the length of the input record.

***** If the logical unit is assigned to the terminal, EMT READ will echo back the read characters onto the printer. An input line will be terminated by entering carriage return [pushing "RETURN" key] , but carriage return is not placed in user buffer. When a new line is to be read, the user is prompted with "?" [it is possible to change this prompt character], then after "READ" operation termination a carriage return and line feed will be supplied automatically, however this happens as a result of echoing back last "RETURN".

Both "BACK SPACE" [to delete the last character read] and "DELETE" [to delete the whole input line] keys on the dec-writer can be used.

NOTE: After issuing an "EMT READ", and before issuing another "EMT READ" to the same device an "EMT WAIT" should be issued first, in order to make sure that the last read operation is completed successfully.

The EMT number for this routine is 10.

Example(s):

```
        MOV   =SCARDS,-(SP)
                          READ FROM
        MOV   =BUF,-(SP)
                          ...SCARDS (UNIT 26)
        EMT   READ        ... FROM BUFFER BUF
        BCS   BADUNIT     IF ERROR
        ..
        MOV   =0,-(SP)    WAIT FOR UNIT 26
        MOV   =X'8400',-(SP) WAIT UNTIL DONE
        EMT   WAIT
        TST   (SP)+       POP UNIT NUMBER
        ADD   PC,@SP      VECTOR TO
        MOV   @(SP)+,PC   PROPER HANDLER
        DC    A"AOK"
        DC    A"EOF"
        DC    A"REC2LONG"
        DC    A"CNTLZ"
        ..
BUF     DS    256C
```

Error(s): If an illegal unit number occurs the C BIT is set to "1".

READB

EMT Routine

Purpose: To read a byte from a logical unit.

Calling parameters:

SP+2
: logical unit number

SP
: address of the byte

Return parameters:

None

Remarks: The logical unit must be between 0 and 30.

Although the routine returns immediately, the transmission is not necessarily complete.

***** NOTE: "EMT READB" will not echo back the entered character, and does not print any prompt character either, it merely reads the entered character, all keys on the DECwriter are treated the same, and will be placed in the user buffer. Thus unlike the "EMT READ", no carriage return, line feed , delete or back space are provided here, user is responsible for such editting procedures.

After issuing an "EMT READB", and before issuing the next "EMT READB" to the same device an "EMT WAIT" should be issued first to make sure that the previous read operation has been completed successfully.

The EMT number for this routine is 12.

Example(s):

```
MOV   =SCARDS,-(SP) READ CHARACTER
MOV   =BUF,-(SP) FROM SCARDS (UNIT 26)
EMT   READB      ... INTO BUF
BCS   BADUNIT    IF ERRORS
..
MOV   =0,-(SP)   WAIT FOR UNIT 26
MOV   =X'8400',-(SP) WAIT UNTIL DONE
EMT   WAIT
TST   (SP)+      POP UNIT NUMBER
ADD   PC,@SP     VECTOR TO
```

```
            MOV   @(SP)+,PC PROPER HANDLER
            DC    A"AOK"
            DC    A"EOF"
            ..
    BUF     DS    1C
```

Error(s): If an illegal unit number occurs the C BIT is set to "1".

READW

EMT Routine

Purpose: To read a word from a logical unit.

Calling parameters:

```
  SP+2
: logical unit number

  SP
: address of the word
```

Return parameters:

None

Remarks: The logical unit must be between 0 and 30.

Although the routine returns immediately, the transmission is not necessarily complete.

After issuing an "EMT READW", and before issuing the next "EMT READW" to the same device an "EMT WAIT" should be issued first to make sure that the previous read operation has been completed successfully.

The EMT number for this routine is 24.

Example(s):

```
        MOV   =10,-(SP) READ WORD
        MOV   =BUF,-(SP) FROM UNIT 10
        EMT   READW     ... INTO BUF
        BCS   BADUNIT   IF ERRORS
        ..
        MOV   =X'0600',-(SP) WAIT FOR UNIT 10
        MOV   =X'8000',-(SP) WAIT UNTIL DONE
        EMT   WAIT
        TST   (SP)+     POP UNIT NUMBER
        ADD   PC,@SP    VECTOR TO
        MOV   @(SP)+,PC PROPER HANDLER
        DC    A"AOK"
        DC    A"EOF"
        ..
BUF     DS    1F
```

Error(s): If an illegal unit number occurs the C BIT is set to "1".

RELBUF

EMT Routine

Purpose: To release a buffer previously obtained through GETBUF.

Calling parameters:

SP
: address of the first word in the buffer to be released

Return parameters:

None

Remarks: The buffer beginning at the given address is released. As the LSI has no memory protect features, the user must be careful not to write over any memory he has not been given by GETBUF, and to return the correct address to RELBUF. OTHERWISE THIS ROUTINE WILL LIKELY CAUSE THE SYSTEM TO BOMB.

The EMT number for this routine is 6.

Example(s):

```
        MOV   STRADR,-(SP)
                        PUSH STARTING ADDRESS
        EMT   RELBUF    RELEASE IT
        ..

STRADR  DS    F         PLACE ADDRESS IS STORED
```

Assuming that a previous GETBUF call has placed an address in STRADR the buffer space so designated will be released.

Error(s): No errors are detected by this routine. However, an attempt to release improper space will WREAK UNKNOWN HAVOC upon things, which may not be detected until after the next call to GETBUF.

RESET

EMT Routine

Purpose: To return all resources to the operating system

calling parameters:

None

Return parameters:

None

Remarks: Program execution is terminated and all memory space is returned to the system. All logical I/O units are reset to *MSOURCE* or *MSINK*. Control is returned to the user rather than to OSWIT, which in most cases is very dangerous. It is therefore recommended that unless the user needs control to be returned to him/her, he/she issue an EMT EXIT rather than EMT RESET.

The EMT number for this routine is 4.

Example(s): EMT RESET

Error(s): None

SCAN

EMT Routine

Purpose: To scan a character string for break characters.

calling parameters:

SP+6
: length of break characters

SP+4
: address of break character

SP+2
: length of string to scan

SP
: address of string to scan

Return parameters:

SP
: address of scanned string

SP+2
: length of scanned string

SP+4
: Return codes:

SP+6
: updated address of string to scan

SP+8
: updated length of string to scan

SP+10
: address of break chars. (unaltered)

SP+12
: length of break chars. (unaltered)

Remarks: The address and length of the break characters are returned unaltered for the next call.

The address and length of the string left to be scanned beyond the break character found are returned on the stack (for the next call).

The EMT number for this routine is 29.

Example(s):

```
            CLR   R0
            MOVB  BREAKS,-(SP) PUSH THE LENGTH AND
            MOV   =BREAKS+1,-(SP) * ADDRESS OF THE BREAK
    *                         * CHARS.
            MOVB  STRING,-(SP) PUSH THE LENGTH AND
            MOV   =STRING+1,-(SP) * ADDRESS OF THE STRING
    *                         * TO SCAN
    SCANMOR EMT   SCAN        SCAN TO A BREAK
            EMT   D2BIN       KEEP A
            ADD   (SP)+,R0    * RUNNING TOTAL
            TST   (SP)+       POP OFF D2BIN PARAMETER
            ADD   PC,@SP      CHECK RETURN
            MOV   @(SP)+,PC * CODE FROM SCAN
            DC    A"Z"        END OF STRING
            DC    A"SCANMOR" :
            DC    A"SCANMOR" ;
            DC    A"SCANMOR" =
    Z       ADD   =8,SP       REMOVE STRING AND
    *                           BREAK CHAR. DESC.
            HALT
    STRING  DC    H"5",C'10:20' <OR> STRING STRCON '10:20'
    BREAKS  DC    H"3",C':;=' <OR> BREAKS STRCON ':;='
```

This program will halt with 30 (decimal) in r0.

Return codes:

2 : string was terminated by an "end of string"

4 : the first break character caused termination

6 : the second break character caused termination

. . .

Error(s): None

START

EMT Routine

Purpose: To schedule a task for immediate execution.

Calling Paramters:

(SP) : task ID

Remarks: This will cause the specified task to be immediatly inserted in the execute queue. If it has a higher priority than the task that scheduled it then it will begin execution, otherwise it will be blocked until it becomes the highest priority task.

The EMT number of this routine is 43.

Example(s):

```
        MOV   ID,-
(SP)            PUSH THE TASK ID
        EMT   START                 START THE TASK
```

Error(s): An error will occur if an illegal ID is passed.

WAIT

EMT Routine

Purpose: To determine the status (whether or not done) of a previous I/O request and/or to wait for the completion of one of a specified set of requests.

Calling parameters:

SP+2
: a word whose bits correspond to the logical units 0 through 15 with bit 0 being unit 0.

SP
: a word whose bits correspond to the logical units 16 through 30 with bit 0 being unit 16.

Return parameters:

If the V BIT is set then:

SP
: the unit number of the I/O task completed.

SP+2
: the return code.

otherwise the stack is clear.

Remarks: If bit 15 of the word on top of the stack is set to 1, the system will wait until one of the logical units corresponding to 1 bits set in the calling parameters has completed its requested I/O task. This unit number will be placed on top of the stack the V BIT set to "1", and a return made to the calling program.

If bit 15 of the word on top of the stack is set to 0, the system checks the completion status of the logical I/O units corresponding to 1 bits set in the calling parameters, and returns without waiting to the calling program. If any of the marked logical I/O units is done, the V BIT is set to "1". The number of that logical I/O unit which is lowest among those done is returned on top of the stack. If none of the indicated logical units is done, the V BIT is cleared.

The EMT number for this routine is 17.

Example(s):

```
WAIT    EQU             DEFINE WAIT
```

```
            MOV   =3,-(SP)  WAIT FOR UNITS 0 OR 1
            MOV   =O'100000',-(SP)
                            SET WAIT BIT
            EMT   WAIT
            MOV   (SP)+,UNITNUM
            ADD   PC,@SP    BRANCH
            MOV   @(SP)+,PC TO CONTINUE
            DC    A"AOK"
            DC    A"EOF"
            DC    A"LONGLINE"
            ..
UNITNUM     DS    F


            MOV   =3,-(SP)  CHECK STATUS
            CLR   -(SP)
            EMT   WAIT      OF 0 AND 1
            BVC   NONEDONE
            MOV   (SP)+,UNITNUM
                            SAVE UNIT
            ADD   PC,@SP
            MOV   @(SP)+,PC PROCEED
            DC    A"AOK"
            DC    A"EOF"
            DC    A"LONGLINE"
            ..
UNITNUM     DS    F
```

Return codes:

```
  2
: successful completion of I/O operation

  4
: end of file on read; end of disk on write;

  6
: line too long (> 255);

  8
:  line  terninated  by control Z if in MTS mode of
     TTY
```

: WHENA and WHENB

EMT Routines

Purpose: To schedule a task when a device interrupt occurs.

Calling Paramters:

(SP)+2 : logical unit number of device

(SP) : task ID

Remarks: These two routines are used to have a task scheduled when the A or B interrupt of a interface card occurs. The unit to have a WHEN condition enabled on must be opened to the appropriate device before the call to WHEN is issued. Also all WHEN's on a device should be cancelled before the device is closed(except when exitting, the WHENs will be cancelled at the same time the device is closed).

The EMT numbers of these routines are 36 and 37 respectively.

Example(s):

```
          MOV   =ASSNTN,-
(SP)       ASSIGN THE TRAIN TO UNIT 5
          EMT   OPEN

          BVS   NOTRAIN              ERROR IF NO TRAI
          MOV   =5,-
(SP)            PUSH UNIT NUMBER
          MOV   ID,-(SP)             PUSH TASK ID
          EMT   WHENA                ASSIGN PHOTO-
CELL TASK

ASSNTN    STRCON '5=*TRAIN*'
```

Error(s): An error will occur if an illegal device is assigned to the unit specified, if an illegal unit number is specified, or if an illegal ID is given.

WRITE

EMT Routine

Purpose: To write a logical record to a device.

Calling parameters:

SP+2
: logical unit number

SP
: the buffer address

Return parameters:

None

Remarks: The logical unit specified must be between 0 and 30.

Although the routine returns immediately, the transmission is not necessarily complete.

The first byte of the buffer must specify the buffer length.

***** NOTE: if the logical unit is assigned to the terminal, "EMT WRITE" will supply a "CARRIAGE RETURN" and a "LINE FEED" after performing a write operation. There is no need to wait for completion of the "WRITE" operation after issuing an "EMT WRITE" and before issuing the next "EMT WRITE", to the same device, as the operating system will provide buffering facilities in this case.

The EMT number for this routine is 11.

Example(s):

```
MOV   =13,-(SP) WRITE
MOV   =TEXT,-(SP) ... TEXT ON
EMT   WRITE     ... UNIT 13
BCS   BADUNIT   IF ERROR
..
MOV   =X'2000',-(SP) WAIT FOR UNIT 13
MOV   =X'8000',-(SP) WAIT UNTIL DONE
EMT   WAIT
TST   (SP)+     POP UNIT NUMBER
ADD   PC,@SP    VECTOR TO
MOV   @(SP)+,PC PROPER HANDLER
DC    A"AOK"
DC    A"DKFULL"
```

```
          ..
TEXT      DC    H"14",C'SAMPLE STRING.'
```

Error(s): If an illegal unit number occurs the C BIT is set to "1".

WRITEB

EMT Routine

Purpose: To write a single byte to a logical unit.

Calling parameters:

SP+2
: logical unit number

SP
: the address of the byte

Return parameters:

None

Remarks: The logical unit specified must be between 0 and 30.

Although the routine returns immediately, the transmission may not be complete.

***** NOTE: "EMT WRITEB" merely prints out the specified character, all codes are valid and therefore, no carriage return or line feed are supplied automatically, the user is responsible for all such editing procedures. There is no need to wait for completion of an "EMT WRITEB" in order to issue the next "EMT WRITEB" to the same device, the operating system provides buffering facilities in this case.

The EMT number for this routine is 13.

Example(s):

```
        MOV   =30,-(SP) WRITE CHARACTER
        MOV   =BYTE,-(SP) ... FROM BUFFER BYTE
        EMT   WRITEB    ... TO UNIT 30 (SERCOM)
        BCS   BADUNIT   IF ERROR
        ..
        CLR   -(SP)     WAIT FOR UNIT 30
        MOV   =X'C000',-(SP) WAIT UNTIL DONE
        EMT   WAIT
        TST   (SP)+     POP UNIT NUMBER
        ADD   PC,@SP    VECTOR TO
        MOV   @(SP)+,PC PROPER HANDLER
        DC    A"AOK"
        DC    A"DKFULL"
        ..
BYTE    DS    1C
```

Error(s): If an illegal unit number occurs the C BIT is set to "1".

WRITEW

EMT Routine

Purpose: To write a word to a logical unit.

Calling parameters:

SP+2
: logical unit number

SP
: address of the word

Return parameters:

None

Remarks: The logical unit must be between 0 and 30.

A wait must be issued to insure that the operation is complete.

The EMT number for this routine is 25.

Example(s):

```
            MOV   =12,-(SP) WRITE WORD
            MOV   =BUF,-(SP) TO UNIT 12
            EMT   WRITEW    ... FROM BUF
            BCS   BADUNIT   IF ERRORS
            ..
            MOV   =X'1000',-(SP) WAIT FOR UNIT 12
            MOV   =X'8000',-(SP) WAIT UNTIL DONE
            EMT   WAIT
            TST   (SP)+     POP UNIT NUMBER
            ADD   PC,@SP    VECTOR TO
            MOV   @(SP)+,PC PROPER HANDLER
            DC    A"AOK"
            DC    A"DKFULL"
            ..
BUF         DS    1F
```

Error(s): If an illegal unit number occurs the C BIT is set to "1".

UNLOCK

EMT Routine

Purpose: To set a locked task back to normal.

Calling Paramters:

None

Remarks: This routine will reset the priority of the active task back to the priority specified in the original task definition. A locked task will be restored to normal. If the active task was not locked then this routine has no effect.

The EMT numberfor this routine is 45.

Example(s):

```
EMT   UNLOCK              RESET THE TASK P
```

Error(s): None

Appendix C: SYSTEM SUBROUTINES AND FUNCTIONS

ATAN

Function Description

Purpose:	To calculate the arc-tangent.
Location:	CRASHLIB
Calling Sequence:	
CRASH:	Y = ATAN (X);
Parameter:	X - Real.
Value returned:	Y - Real, value in radians.
Routines used:	#POLY, #FCMP
Author:	John J. Puttress
Last update:	April 21, 1976

COS

Function Description

Purpose:	To calculate the cosine.
Location:	CRASHLIB
Calling Sequence:	
CRASH:	Y = COS (X);
Parameter:	X - Real, value in radians
Value returned:	Y - Real
Routines used:	#POLY
Author:	John J. Puttress
Last update:	April 21, 1976

D2FLOAT

Subroutine Description

Purpose: To convert a string of ASCII characters into a floating point value.

Location: CRASHLIB

Calling Sequence:

Assembly:
```
CALL D2FLOAT,(BUFFER,LENGTH,FLOATING,
     INDEX,RCODE)
```

Parameters:

BUFFER - First byte of data to be converted.

LENGTH - A fullword location containing the length of the buffer to convert.

FLOATING - A double word location to place the floating result.

INDEX - A fullword location to return the address of the next byte in the buffer to scan. Points to break character or byte after last buffer location.

RCODE - A fullword location to return the error code.

Return codes:

0 - ok
2 - exponent overflow underflow
4 - conversion error

Descritpion: This subroutine scans the input string for three cases:

```
XXXXX
X.XXXX
and X.XXXXEXX
```

including sign for both mantisa and exponent. It then converts it into a floating point number. the index returned points to either the character that stopped the conversion (non-numeric) or the byte after the last buffer location (end of field).

Routines used: #SCALE,#SCALE2

Author: John J. Puttress

Last update: April 21, 1976

DOPEFIX

Subroutine Description

Purpose:	An intrinsic CRASH subroutine to setup an automatic dope
Location:	CRASHLIB vector depending on passed parameters.
Calling Sequence:	
Assembly:	CALL DOPEFIX,(ADOPE)
Parameter:	ADOPE - Array's dope vector.
Descritpion:	This routine takes the dope vector shell and fills in missing information. the element size, number of dimensions, lower bounds and upper bounds (passed in size) are filled in at call. The (virtual base - base address), size and scale factor are filled in on return. Also on return, in register 0, the number of bytes required to create the array.
Routines used:	NONE
Author:	John J. Puttress
Last update:	April 21, 1976

EXP

Function Description

Purpose:	To calculate the exponential.
Location:	CRASHLIB
Calling Sequence:	
CRASH:	Y = EXP (X);
Parameter:	X - Real, where -88.02 < X < 88.02
Value returned:	Y - Real
Routines used:	#FCMP, #IFIX, #FLOAT
Author:	John J. Puttress
Last update:	April 21, 1976

1. ON ERROR, V-BIT IS SET, CLEARED OTHERWISE.

FLOAT2D

Subroutine Description

Purpose: To convert a floating point number into a character string.

Location: CRASHLIB

Calling Sequence:

Assembly: CALL FLOAT2D,(FLOATING,BUFFER)

Parameters: BUFFER - First byte of a buffer to place resultant character string. this buffer must be at least 11 bytes long.

FLOATING - A double word containing the floating point number to be converted.

Descritpion: This routine converts the floating point number into one of two 11 byte formats:

'.XXXXXEXX'
or ' .XXXXEXX'

If the number is positive, the sign is suppressed (the exponent always has a sign). if there is an overflow underflow, the buffer is filled with '*'.

Routines used: #SCALE,#SCALE2

Author: John J. Puttress

Last update: April 21, 1976

LOG

Function Description

Purpose:	To calculate the logarithm.
Location:	CRASHLIB
Calling Sequence:	
CRASH:	Y = LOG (X);
Parameter:	X - Real, where x 0
Value returned:	Y - Real
Routines used:	#FCMP, #FLOAT, #POLY
Author:	John J. Puttress
Last update:	April 21, 1976

1. on error, V-bit is set, cleared otherwise.

SIN

Function Description

Purpose:	To calculate the sine.
Location:	CRASHLIB
Calling Sequence:	
CRASH:	Y = SIN (X);
Parameter:	X - Real, value in radians
Value returned:	Y - Real
Routines used:	#POLY
Author:	John J. Puttress
Last update:	April 21, 1976

SQRT

Function Description

Purpose:	To calculate the square root.
Location:	CRASHLIB
Calling Sequence:	
CRASH:	Y = SQRT (X);
Parameter:	X - Real, where X > 0
Value returned:	Y - Real (Note: if X is negitive, the SQRT of the absolute value is taken and its negitive value is returned).
Routines used:	NONE
Author:	John J. Puttress
Last update:	April 21, 1976

#BMTXMUL

Subroutine Description

Purpose:	An intrinsic CRASH routine to multiply two boolean arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #BMTXMUL,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Boolean array dope vector.
	BDOPE - Boolean array dope vector.
	CDOPE - Boolean array dope vector (product).
Operation:	C = A * B;
Routines used:	NONE
Author:	John J. Puttress
Last update:	April 21, 1976

#FCMP

Routine Description

Purpose:	To compare two floating point number and set appropriate condition codes.
Location:	CRASHLIB
On call:	R0 A(Source) R1 A(Destination)
Operation:	(Source) - (Destination)
Condition codes:	N: set if Result < 0, cleared otherwise Z: set if Result = 0, cleared otherwise V: cleared C: cleared
Routines used:	NONE
Author:	John J. Puttress
Last update:	April 21, 1976

#FLOAT

Subroutine Description

Purpose:	To convert an integer into a floating point number.
Location:	CRASHLIB
On call:	SP PC SP+2 Integer value
On return:	SP Floating Hi SP+2 Floating Lo Registers 0 and 1 modified.
Routines used:	NONE
Author:	John J. Puttress
Last update:	April 21, 1976

#FMTX2I

Subroutine Description

Purpose: An intrinsic CRASH subroutine to convert a real array into an integer array.

Location: CRASHLIB

Calling Sequence:

Assembly: CALL #FMTX2I,(ADOPE,BDOPE)

Parameters: ADOPE - Real array's dope vector (source array).

BDOPE - Integer array's dope vector (destination array).

Operation: B = A;

Routines used: #IFIX

Author: Rick Richardson

Last update: September 1, 1978

#FMTXADD

Subroutine Description

Purpose:	An intrinsic CRASH routine to add two real arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #FMTXADD,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Real array dope vector.
	BDOPE - Real array dope vector.
	CDOPE - Real array dope vector (sum).
Operation:	C =A + B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#FMTXMUL

Subroutine Description

Purpose:	An intrinsic CRASH routine to multiply two real arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #FMTXMUL,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Real array dope vector. BDOPE - Real array dope vector. CDOPE - Real array dope vector (product).
Operation:	C = A * B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#FMTXSUB

Subroutine Description

Purpose:	An intrinsic CRASH routine to subtract two real arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #FMTXSUB,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Real array dope vector.
	BDOPE - Real array dope vector.
	CDOPE - Real array dope vector (difference).
Operation:	C = A - B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#FSCLDIV

Subroutine Description

Purpose:	An intrinsic CRASH routine to divide a real array by a scaler.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #FSCLDIV,(ASCALER,BDOPE,CDOPE)
Parameters:	ASCALER - Real scaler.
	BDOPE - Real array dope vector.
	CDOPE - Real array dope vector (quotient).
Operation:	C = B A;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#FSCLMUL

Subroutine Description

Purpose:	An intrinsic CRASH routine to multiply a real array by a scaler.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #FSCLMUL,(ASCALER,BDOPE,CDOPE)
Parameters:	ASCALER - Real scaler.
	BDOPE - Real array dope vector.
	CDOPE - Real array dope vector (product).
Operation:	C = B * A;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IFIX

Routine Description

Purpose:	To convert a floating point number into an integer
Location:	CRASHLIB (the fraction is truncated).
On call:	SP PC SP+2 Floating Hi SP+4 Floating Lo
On return:	SP Integer value Registers 0 and 1 modified.
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IMTX2F

Subroutine Description

Purpose:	An intrinsic CRASH subroutine to convert an integer array into a real array.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IMTX2F,(ADOPE,BDOPE)
Parameters:	ADOPE - Integer array's dope vector (source array). BDOPE - Real array's dope vector (destination array).
Operation:	B = A;
Routines used:	#FLOAT
Author:	Rick Richardson
Last update:	September 1, 1978

#IMTXADD

Subroutine Description

Purpose:	An intrinsic CRASH routine to add two integer arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IMTXADD,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Integer array dope vector.
	BDOPE - Integer array dope vector.
	CDOPE - Integer array dope vector (sum).
Operation:	C = A + B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IMTXAND

Subroutine Description

Purpose:	An intrinsic CRASH routine to and two integer arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IMTXAND,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Integer array dope vector.
	BDOPE - Integer array dope vector.
	CDOPE - Integer array dope vector (result).
Operation:	C = A AND B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IMTXMUL

Subroutine Description

Purpose:	An intrinsic CRASH routine to multiply two integer arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IMTXMUL,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Integer array dope vector.
	BDOPE - Integer array dope vector.
	CDOPE - Integer array dope vector (product).
Operation:	C = A * B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IMTXOR

Subroutine Description

Purpose:	An intrinsic CRASH routine to or two integer arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IMTXOR,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Integer array dope vector.
	BDOPE - Integer array dope vector.
	CDOPE - Integer array dope vector.
Operation:	C = A OR B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IMTXSUB

Subroutine Description

Purpose:	An intrinsic CRASH routine to subtract two integer arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IMTXSUB,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Integer array dope vector. BDOPE - Integer array dope vector. CDOPE - Integer array dope vector (difference).
Operation:	C = A - B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IMTXXOR

Subroutine Description

Purpose:	An intrinsic CRASH routine to exclusive or two integer arrays.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IMTXXOR,(ADOPE,BDOPE,CDOPE)
Parameters:	ADOPE - Integer array dope vector.
	BDOPE - Integer array dope vector.
	CDOPE - Integer array dope vector.
Operation:	C = A XOR B;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#ISCLDIV

Subroutine Description

Purpose:	An intrinsic CRASH routine to divide an integer array by a scaler.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #ISCLDIV,(ASCALE,BDOPE,CDOPE)
Parameters:	ASCALE - Integer scaler.
	BDOPE - Integer array dope vector.
	CDOPE - Integer array dope vector (quotient).
Operation:	C = B A;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#ISCLMUL

Subroutine Description

Purpose:	An intrinsic CRASH routine to multiply an integer array by a scaler.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #ISCLMUL,(ASCALE,BDOPE,CDOPE)
Parameters:	ASCALE - Integer scaler.
	BDOPE - Integer array dope vector.
	CDOPE - Integer array dope vector (product).
Operation:	C = B * A;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IROUND

Routine Description

Purpose:	To convert a floating point number into an integer (the fraction is rounded up if .5).
Location:	CRASHLIB
On call:	SP PC SP+2 Floating Hi SP+4 Floating Lo
On return:	SP Integer value Registers 0 and 1 modified.
Routines used:	NONE
Author:	John J. Puttress
Last update:	April 21, 1976

#MTXMOV

Subroutine Description

Purpose:	An intrinsic CRASH routine to move the contents of one array to another.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #MTXMOV,(ADOPE,BDOPE)
Parameters:	ADOPE - Array dope vector (source).
	BDOPE - Array dope vector (destination).
Operation:	B = A;
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#POLY

Routine Description

Purpose:	To evaluate a polynomial (floating point).
Location:	CRASHLIB
On call:	R0 A(X) R1 A(X Increment) R2 A(Coefficients) R3 # of terms to calculate.
On return:	SP Result Hi SP+2 Result Lo Registers 3 and 5 modified.
Routines used:	NONE
Author:	John J. Puttress
Last update:	April 21, 1976

#POWER

Routine Description

Purpose:	Internal routine used by #POWERII, #POWERRI and #POWERRR to calculate: Base ** Exponent where Base and Exponent are both real.
Location:	CRASHLIB
On call:	SP PC SP+2 Base Hi SP+4 Base Lo SP+6 Exponent Hi SP+8 Exponent Lo
On return:	SP Result Hi SP+2 Result Lo Registers 0, 1, 2 and 5 modified.
Descritpion:	This routine assumes that all parameters are good. the result is calculated using the following equation: EXP (Y * LOG (X))
Routines used:	LOG, EXP
Author:	John J. Puttress
Last update:	April 21, 1976

1. on error, the V-bit is set, cleared otherwise.

#POWERII

Routine Description

Purpose:	To perform the intrinsic CRASH function: INTEGER ** INTEGER
Location:	CRASHLIB
On call:	SP PC SP+2 Exponent SP+4 Base
On return:	SP result
Routines used:	#POWER, #IFIX, #FLOAT
Author:	John J. Puttress
Last update:	April 21, 1976

1. on error, Result=0 and V-bit is set, V-bit cleared otherwise.

#POWERRI

Routine Description

Purpose:	To perform the intrinsic CRASH function: REAL ** INTEGER
Location:	CRASHLIB
On call:	SP PC SP+2 Exponent SP+4 Base Hi SP+6 Base Lo
On return:	SP Result Hi SP+2 Result Lo
Routines used:	#POWER, #FLOAT
Author:	John J. Puttress
Last update:	April 21, 1976

1. on error, Result=0 and V-bit is set, V-bit cleared otherwise.

#POWERRR

Routine Description

Purpose: To perform the intrinsic CRASH function:

REAL ** REAL

Location: CRASHLIB

On call:

SP	PC
SP+2	Exponent Hi
SP+4	Exponent Lo
SP+6	Base Hi
SP+8	Base Lo

On return:

SP	Result Hi
SP+2	Result Lo

Routines used: #POWER

Author: John J. Puttress

Last update: April 21, 1976

1. on error, Result=0 and V-bit is set, V-bit cleared otherwise.

#SCALE

Routine Description

Entry: #SCALE2

Purpose: An internal routine used by D2FLOAT and FLOAT2D to scale a floating point number by a power of 10.

Location: CRASHLIB

On call:
```
SP    R5
SP+2  Hi Float
SP+4  Lo Float
[R3 | R4 Power of 10 for #SCALE | #SCALE2]
```

On return:
```
SP    Scaled Float Hi
SP+2  Scaled Float Lo
Registers 3, 4 and 5 modified.
All other registers unchanged.
```

Descritpion: This routine performs the scaling in two steps:

#SCALE - scales by 10**1, 10**2, ..., 10**7

#SCALE2 - scales by 10**-40, 10**-32, ..., 10**32, 10**40

These operations must be none in two steps since scaling on input and output presents different exponent overflow underflow problems.

Routines used: NONE

Author: John J. Puttress

Last update: April 21, 1976

1. if the V-bit is set on return, a scaling error has occured, the stack has been cleared. V-bit cleared otherwise.

#SUBSCR

Function Description

Entry: #SUBSCR1

Purpose: To calculate the array element address for a given array and subscripts.

Location: CRASHLIB

Calling Sequence:

Assembly:
```
CALL #SUBSCR,(ADOPE,BSUBSCR)

     CALL #SUBSCR1,(ADOPE,BSUBSCR,CERROUT,DEF
```

Parameters:

ADOPE - Array's dope vector.

BSUBSCR - Subscript list.

CERROUT - Error handler.

DEFAULT - Default value in case of error.

Value returned: Address of array element.

Descritpion: This routine calculates the address of an array element. If the entry point #SUBSCR1 is used, subscript checking is done. if there is an error, the error handler is called and the default value is returned. Entry point #SUBSCR ignores all errors.

Routines used: NONE

Author: John J. Puttress

Last update: April 21, 1976

#CATNATE

Function Description

Purpose: To concatenate several strings into one larger string. Automatic conversion of Real and Integer numbers is performed.

Location: CRASHLIB

Calling Sequence:

Assembly: CALL #CATNATE,(TYPE1,ADDR1,...,TYPEN,ADDRN)

Parameters:

TYPE - 0 If Character
1 If Integer
-1 If Real.

ADDR - Address of string or number.

Value returned: R1 points to new string. Must issue a $RELBUF to return string to OSWIT.

Descritpion: The list of variables is scanned. If conversion is indicated, the number is converted into ASCII characters. The string is built up in a 258 byte buffer from OSWIT. Hence, it must be released when it is no longer needed. An EMT $ERROR occurs if concatenation results in a string longer than 255 characters.

Routines used: $BIN2D, FLOAT2D, $ERROR, $GETBUF, $RELBUF

Author: Rick Richardson

Last update: September 1, 1978

#SUBSTR

Routine Description

Purpose:	An intrinsic CRASH routine to get a substring of a character string.
Location:	CRASHLIB
On call:	SP PC SP+2 Address of string SP+4 Number of characters SP+6 Number of the starting character in string
On return:	SP Address of an OSWIT buffer containing the substring
Routines used:	$GETBUF
Author:	Rick Richardson
Last update:	September 1, 1978

#BITSEL

Routine Description

Purpose:	An intrinsic CRASH routine to get bit strings from an integer or bit variable.
On call:	SP PC SP+2 Address of bits to be selected SP+4 Number of bits to take SP+6 Number of the starting bit (As DEC numbers them)
On return:	SP The selected bits, right justified in the word.
Routines used:	NONE
Author:	Rick Richardson
Last update:	September 1, 1978

#IR

Subroutine Description

Purpose:	To INPUT a real variable from SCARDS.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IR,(REAL.VARIABLE)
Parameters:	REAL.VARIABLE - Where you wish to have the converted value placed.
Descritpion:	This subroutine will scan the input stream SCARDS and convert from that source one real variable.
Routines used:	#READ,D2FLOAT,#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#IRD

Subroutine Description

Purpose:	To INPUT a real analog variable from SCARDS applying OFFSET and SCALE factor.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IRD,(REAL.VARIABLE,DOPE.VECTOR)
Parameters:	REAL.VARIABLE - Where you wish to have the converted value placed.
	DOPE.VECTOR - Dope vector of the analog variable.
Descritpion:	This subroutine will scan the input stream SCARDS and convert from that source one real variable then it will apply the OFFSET and SCALE called for in the dope vector.
Routines used:	#READ,D2FLOAT,#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#II

Subroutine Description

Purpose:	To INPUT an integer variable from SCARDS.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #II,(INTEGER.VARIABLE)
Parameters:	INTEGER.VARIABLE - Where you wish to have the converted value placed.
Descritpion:	This subroutine will scan the input stream SCARDS and convert from that source one integer variable.
Routines used:	#READ,#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#IID

Subroutine Description

Purpose: To INPUT an integer analog variable from SCARDS applying OFFSET and SCALE factor.

Location: CRASHLIB

Calling Sequence:

Assembly: CALL #IID,(INTEGER.VARIABLE,DOPE.VECTOR)

Parameters: INTEGER.VARIABLE - Where you wish to have the converted value placed.

DOPE.VECTOR - Dope vector of the analog variable.

Descritpion: This subroutine will scan the input stream SCARDS and convert from that source one integer variable then it will apply the OFFSET and SCALE called for in the dope vector.

Routines used: #READ,#FLOAT,#IFIX,#DUMMY1

Author: Ted J. Kowalski

Last update: June 23, 1976

#IS

Subroutine Description

Purpose:	To INPUT a string variable from SCARDS.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #IS,(STRING.VARIABLE)
Parameters:	STRING.VARIABLE - Where you wish to have the string placed.
Descritpion:	This subroutine will scan the input stream SCARDS and get from that source one string variable.
Routines used:	#READ,#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#READ

Subroutine Description

Purpose:	To read a string from SCARDS placing that string in the crash INPUT/OUTPUT buffer.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #READ
Descritpion:	This subroutine will scan the input stream SCARDS and get from that source one line for the other INPUT routines.
Routines used:	#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#OR

Subroutine Description

Purpose:	To OUTPUT a real variable to SPRINT.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #OR,(REAL.VARIABLE)
Parameters:	REAL.VARIABLE - The real variable you wish to have OUTPUT.
Descritpion:	This subroutine will convert one real variable and place that result on the output stream SPRINT.
Routines used:	#WRITE,D2FLOAT,#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#ORD

Subroutine Description

Purpose:	To OUTPUT a real analog variable to SPRINT applying OFFSET and SCALE factor.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #ORD,(REAL.VARIABLE,DOPE.VECTOR)
Parameters:	REAL.VARIABLE - The real variable you wish to have OUTPUT.
	DOPE.VECTOR - Dope vector of the analog variable.
Descritpion:	This subroutine will convert one real variable applying OFFSET and SCALE factors and will place that result on the output stream SPRINT.
Routines used:	#WRITE,D2FLOAT,#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#OI

Subroutine Description

Purpose:	To OUTPUT an integer variable to SPRINT.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #OI,(INTEGER.VARIABLE)
Parameters:	INTEGER.VARIABLE - The integer variable you wish to have OUTPUT.
Descritpion:	This subroutine will convert one integer variable and place that result on the output stream SPRINT.
Routines used:	#WRITE,#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#OID

Subroutine Description

Purpose:	To OUTPUT an integer analog variable to SPRINT applying OFFSET and SCALE factor.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #OID,(INTEGER.VARIABLE,DOPE.VECTOR)
Parameters:	INTEGER.VARIABLE - The integer variable you wish to have OUTPUT. DOPE.VECTOR - Dope vector of the analog variable.
Descritpion:	This subroutine will convert one integer variable applying OFFSET and SCALE factors and will place that result on the output stream SPRINT.
Routines used:	#WRITE,#FLOAT,#IFIX,#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#OS

Subroutine Description

Purpose: To OUTPUT a string variable to SPRINT.

Location: CRASHLIB

Calling Sequence:

Assembly: CALL #OS,(STRING.VARIABLE)

Parameters: STRING.VARIABLE - The string you wish to OUTPUT.

Descritpion: This subroutine will take a string from the user and place that string on the output stream SPRINT.

Routines used: #WRITE,#DUMMY1

Author: Ted J. Kowalski

Last update: June 23, 1976

#WRITE

Subroutine Description

Purpose:	To write a string from the CRASH INPUT/OUTPUT buffer to SPRINT.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #WRITE
Descritpion:	This subroutine will take the converted string in the CRASH INPUT/OUTPUT buffer and will place that string on the output stream SPRINT.
Routines used:	#DUMMY1
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#DUMMY1

Routine Description

Purpose: INPUT/OUTPUT temporary storage CSECT.

Location: CRASHLIB

Entry:

#BUFFER - The INPUT/OUTPUT string storage area (256 bytes) .

#LENGTH - The length of the input string.

#RNTCDE1 - The return code from the WAIT EMT.

#BRKCHR - The address of the character that caused conversion to stop.

#TMSP - The most significant past of a temporary floating point number.

#TLSP - The least significant part of a temporary floating point number.

Descritpion: This CSECT contains all the necessary storage for INPUT/OUTPUT processing of string data.

Author: Ted J. Kowalski

Last update: June 23, 1976

#GRD

Subroutine Description

Purpose: To GET a real analog variable from a given LDN applying OFFSET and SCALE factor.

Location: CRASHLIB

Calling Sequence:

Assembly: CALL #GRD,(REAL.VARIABLE,DOPE.VECTOR)

Parameters: REAL.VARIABLE - Where you wish to have the converted value placed.

DOPE.VECTOR - Dope vector of the analog variable.

Descritpion: This subroutine will GET from the LDN specified in the dope vector one analog variable then it will apply the OFFSET and SCALE called for in the dope vector.

Routines used: #READG,#FLOAT,#DUMMY2

Author: Ted J. Kowalski

Last update: June 23, 1976

#GID

Subroutine Description

Purpose:	To GET an integer analog variable from a given LDN applying OFFSET and SCALE factor.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #GID,(INTEGER.VARIABLE,DOPE.VECTOR)
Parameters:	INTEGER.VARIABLE - Where you wish to have the converted value placed. DOPE.VECTOR - Dope vector of the analog variable.
Descritpion:	This subroutine will GET from the LDN specified in the dope vector one analog variable then it will apply the OFFSET and SCALE called for in the dope vector.
Routines used:	#READG,#FLOAT,#IFIX,#DUMMY2
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#READG

Subroutine Description

Purpose:	To read a byte from a given LDN placing that byte in the crash GET/PUT buffer.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #READG
Descritpion:	This subroutine will GET from the LDN specified in the dope vector one byte for the other GET routines.
Routines used:	#DUMMY2
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#PRD

Subroutine Description

Purpose: To PUT a real analog variable to a given LDN applying OFFSET and SCALE factor.

Location: CRASHLIB

Calling Sequence:

Assembly: CALL #PRD,(REAL.VARIABLE,DOPE.VECTOR)

Parameters: REAL.VARIABLE - The real variable you wish to have PUT.

DOPE.VECTOR - Dope vector of the analog variable.

Descritpion: This subroutine will apply OFFSET and SCALE factors to the analog variable and will place that result on a given LDN specified by the dope vector.

Routines used: #WRITEP,#IFIX,#DUMMY2

Author: Ted J. Kowalski

Last update: June 23, 1976

#PID

Subroutine Description

Purpose:	To PUT an integer analog variable to a given LDN applying OFFSET and SCALE factor.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #PID,(INTEGER.VARIABLE,DOPE.VECTOR)
Parameters:	INTEGER.VARIABLE - The integer variable you wish to have PUT.
	DOPE.VECTOR - Dope vector of the analog variable.
Descritpion:	This subroutine will apply OFFSET and SCALE factors to the analog variable and will place that result on a given LDN specified by the dope vector.
Routines used:	#WRITEP,#FLOAT,#IFIX,#DUMMY2
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#WRITEP

Subroutine Description

Purpose:	To write a string from the CRASH GET/PUT buffer to a given LDN.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	CALL #WRITEP
Descritpion:	This subroutine will take the scaled byte in the CRASH GET/PUT buffer and will place that byte on the on a given LDN specified by the dope vector.
Routines used:	#DUMMY2
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#DUMMY2

Routine Description

Purpose:	GET/PUT temporary storage CSECT.
Location:	CRASHLIB
Entry:	#TBYTE - byte used for temporary A/D D/A storage. #RTNCDE2 - Return code from the WAIT EMT for GET/PUT operations.
Descritpion:	This CSECT contains all the necessary storage for GET/PUT processing of analog data.
Author:	Ted J. Kowalski
Last update:	June 23, 1976

#GCNV

Subroutine Description

Purpose:	TO APPLY SCALE AND OFFSET WHEN INPUTTING OR GET'TING a variable.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	JSR PC,#GCNV
Descritpion:	THIS SUBROUTINE WILL TAKE THE SCALE AND OFFSET FROM the dope vector whose address is assumed in R0. It will apply them to the REAL VARIABLE on the stack.
Author:	Brian S. Cashman
Last update:	July 18,1976

#PCNV

Subroutine Description

Purpose:	TO APPLY SCALE AND OFFSET WHEN OUTPUTTING OR PUT'TING a variable.
Location:	CRASHLIB
Calling Sequence:	
Assembly:	JSR PC,#PCNV
Descritpion:	THIS SUBROUTINE WILL TAKE THE SCALE AND OFFSET FROM THE DOPE VECTOR WHOSE ADDRESS IS ASSUMED IN R0. IT WILL APPLY them to the REAL VARIABLE on the stack.
Author:	Brian S. Cashman
Last update:	July 18,1976

Appendix D: ODT - ONLINE DEBUGGING TOOL

ODT is a ROM resident microcode debugging package that is located on the LSI-11 CPU board. ODT commands are executed by the LSI-11 processor only when the system is in the HALT mode. When in this mode the processor prompts with an "@" and responds to commands and information entered via the console. All processor responses are controlled by the processor microcode.

All commands and characters are echoed by the processor. Illegal commands are echoed and followed by "?". This result also occurs when no location is open or openning non-existent locations. The console always prints six numeric characters as addresses or data. The user is no t required to enter leading zeros. All input and output addresses and data are in octal.

The following is a list of ODT commands and a brief description of how they are used. For a more detailed explanation and examples see DEC's microcomputer processor manual.

RETURN	Close opened location and accept next command.
LINE FEED	Close current location; open next sequential location.
]	Open previous location.
_	Take contents of opened location, index by opened location plus 2, and open that location.
@	Take contents of opened location as an absolute address and open that location.
r/	Open location r.
/	Open last location.
$n or Rn	Open general register n(0-7) or S (PS register).
r;G or rG	Go to location r, initialize the bus, and start program.
nL	Execute bootstrap loader using n as device CSR address.
;P or P	Proceed with program execution.
RUBOUT OR DELETE	Erase previous character. Response is a backslash \ (134) each time

RUBOUT is entered.

M — Maintance. Display of an internal CPU register follows the M command. Only the last digit is significant, indicating how the cpu entered the HALT (ODT) mode, as follows:

- 0 or 4 HALT instruction or BHALT L bus signal asserted.
- 1 or 5 Bus error occured while getting device interrupt vector.
- 2 or 6 Bus error occuered while doing memory refresh.
- 3 Double bus error occured (stack was non-existent value).
- 4 Reserved instruction trap occurred (non-existent Micro-PC address occurred on internal CPU bus).
- 7 A combination of 1, 2, and 4 occurred.

CRLT-SHFT-S — For manufacturing tests only. Escape this command function by typing NULL and 000 and 100).

Appendix E: ASSEMBLY DEBUG MODE

This is the LSI 11 high-core DEBUG program. It is a core resident program intended for use in initial checkout of software for the LSI mini-computer by direct interaction from the operator's console. Once the software is "up", this program will be used only when all else fails, i.e., if the software fails on a frequent basis. This program runs with interrupts disabled so it cannot be used for debugging while the system is "up".

7.5 General Concepts

The LSI 11 DEBUG routine herein described is a high-core resident utility which processes traps to the low-core vectors at 4, 10, 14, and 20 (hexadecimal). Traps to 4 and 10 are recognized as program interrupts. A trap to 14 may be recognized as the execution of an instruction at a breakpoint causing the breakpoint to be restored. Otherwise, the trap to 14 is ignored. The trap to 20 is recognized as a breakpoint, either preset or set by DEBUG. After any of these traps but 14 (which is transitory), the registers (0-7) and the PS are saved and a message printed indicating how the trap was recognized. Commands may then be entered to display or alter core, set breakpoints, clear all breakpoints, display or alter the saved registers, or restart execution of the program. When the program is restarted, all registers and the PS are restored from the saved registers.

Commands to the DEBUG routine are recognized by their first and last letters. For example, the CONTINUE command could be given as CONTINUE, CTE, CXXBQE, or CE. Parameters to the commands must be separated from each other and the command itself by one or more blanks. All parameters are numbers. Any number may be expressed as the sum of two or more numbers. All commands are executed as they are typed in. Thus, although a line may be deleted, that part of the line up to the first blank will have already been executed. For example, if one typed in 'ALTER 6 0 77 20' where is a rubout (the delete line character), locations 6 and 10 would be altered. Location 12 would not be. The rubout is mainly useful for suppressing the "Eh?" message when you know you have typed in an illegal command.

Illegal commands to the system or missing parameters generate the only error message: "Eh?". There are no illegal parameters per se. Some parameters, however, even though syntactically correct may cause program interrupts within the DEBUG routine. Such things as byte addresses for ALTER or undefined addresses for ALTER, BREAKPOINT, or DISPLAY will cause interrupts. These interrupts will not modify the saved registers though.

The DEBUG routine may be entered directly by starting it from ODT or more commonly by using the OSWIT DEBUG command. Once entered, the following commands can be used:

ALTER {location} {value} [{value}]

ALTER modifies core.

AT {routine}{location} [{location}]

AT sets AT point(s)

BREAK {location} [{location}]

BREAK sets breakpoint(s).

BINFO

BINFO displays breakpoint(s).

CONTINUE

CONTINUE restarts execution after a breakpoint.

CLEAR

CLEAR restores all breakpoints.

CSECT {location}

CSECT sets relative offset.

DISPLAY {location} [{words}]

DISPLAY displays core.

DISPLAYB {location} [{bytes}]

DISPLAYB displays core bytes.

DISPLAY {location} [{words}]

DISPLAY displays core.

GOTO {location}

GOTO restarts execution at a given location.

GR [{register}]

GR displays register(s).

MODGR {register} {value} [{value}]

MODGR modifies register(s).

OSWIT

OSWIT warmstarts the operating system.

STEP [{steps}]

STEP steps through the program.

These commands are described in detail in the section that follows.

7.6 DEBUG command descriptions

ALTER

Command Description

Purpose: To alter the contents of core.

Prototype: ALTER {location} {value} [{value}]

Parameters:

location is the address of a word of PDP11 memory or of a device register.

value is the value that word is to be modified to.

Usage: This command will modify the specified location(s) to the specified value(s). If an odd byte address is specified or an illegal address, a program interrupt will occur within the DEBUG routine.

Example(s):
```
ALTER 0 0 0
AR 202+4000 12366
AR +500 240
```

AT

Command Description

Purpose: To set AT point(s).

Prototype: AT {routine} {location} [{location}]

Parameters:

routine is the address of the routine branched to when the AT point is reached. This routine should terminate with an RTT to return to the interrupted program. The interrupted programs PC & PSW will be on the stack as if an interrupt had vectored directly to the routine. Although, the PSW will have the T bit set by the DEBUG routines. This enables the IOT to be replaced after the interrupted instruction is executed. To enter DEBUG from the routine, an IOT should be executed. No breakpoints or AT points should be set in a AT routine.

location is the address of a word of PDP11 memory at which a breakpoint is to be inserted. It should be the first word of an instruction.

Usage: This command will modify the location(s) specified to X'0004' (the IOT instruction) and save the previous value(s) and the location(s) on the "breakpoint stack". When the IOT instruction is executed, the "breakpoint routine" is trapped to. If the location is found in the "breakpoint stack" and there is a AT routine address for that location, a legitimate AT point is recognized. The CLEAR instruction will restore the location but not affect the saved PC.

Example(s): AT 302 +250
AT 1020+76+16 1344

BREAK

Command Description

Purpose: To set breakpoint(s).

Prototype: BREAK {location} [{location}]

Parameters:

location is the address of a word of PDP11 memory at which a breakpoint is to be inserted. It should be the first word of an instruction.

Usage: This command will modify the location(s) specified to X'0004' (the IOT instruction) and save the previous value(s) and the location(s) on the "breakpoint stack". When the IOT instruction is executed, the "breakpoint routine" is trapped to. If the location is found in the "breakpoint stack", a legitimate breakpoint is recognized. Otherwise, the IOT is assumed to signal a "preset" breakpoint. The CLEAR instruction will restore the location but not affect the saved PC.

J

Example(s): BREAK 302 +250
BK 1020+76+16

BINFO

Command Description

Purpose: To display the locations and old values replaced by breakpoints.

Prototype: BINFO

Usage: The breakpoint locations are displayed, one per line. If a CSECT has been issued, both the absolute and relative addresses are displayed. An at-sign will follow if this is the current breakpoint. The instruction which occupied this location will be displayed next. Finally, if this is a AT point, the AT routine location will be displayed.

Example(s):
```
BINFO
BREAKPOINTINFO
BO
```

CONTINUE

Command Description

Purpose: To continue execution of a program after a breakpoint or just to continue it.

Prototype: CONTINUE

Usage: This command will restart execution of the program. If at a breakpoint which has not been restored, the location is modified back to the original instruction, executed, and re-modified back to an IOT, with execution continuing after this. The trace facility is utilized to do this. Please note that the trace bit will be on in the PS while the instruction at the breakpoint is being executed. If the program is not at a legitimate breakpoint, this command will cause execution to resume at the location specified by the saved PC.

Example(s): CONTINUE
CE

CLEAR

Command Description

Purpose: To restore all breakpoints currently set.

Prototype: CLEAR

Usage: This instruction will restore all breakpoints in the program. It may be given at any time, even at a breakpoint, which then ceases to be a breakpoint.

Example(s):
```
CLEAR
CR
```

CSECT

Command Description

Purpose: To set a displacement to be added to any octal number starting with a "+".

Prototype: CSECT [{location}]

Usage: This controls the address assumed to be the start of the CSECT being debugged. Relative addresses start w a "+". When {location} isn't specified, absolute addressing is assumed.

Example(s): CSECT (resets to zero)
CSECT 10000

DISPLAY

Command Description

Purpose: To display the contents of core.

Prototype: DISPLAY {location} [{words}]

Parameters:

location is the address of a byte of PDP11 memory or of a device register.

words is the number of successive words to be displayed. If left out, it defaults to 1.

Usage: The specified number of locations are displayed, eight per line. If a byte address is given, it is aligned to a word boundary. If an illegal address is specified, a program interrupt will occur within the DEBUG routine.

Example(s):
```
DISPLAY 200 14
DY +100+20 4
DY 177560
```

DISPLAYB

Command Description

Purpose: To display the contents of core in byte form.

Prototype: DISPLAYB {location} [{bytes}]

Parameters:

location is the address of a byte of PDP11 memory or of a device register.

bytes is the number of successive bytes to be displayed. If left out, it defaults to 1.

Usage: The specified number of locations are displayed, sixteen per line. If an illegal address is specified, a program interrupt will occur within the DEBUG routine.

Example(s): DISPLAYB 201 14

DB +102 4
DB 177560

GOTO

Command Description

Purpose: To restart execution of the program at a specified location.

Prototype: GOTO {location}

Parameters:

location is the address of a word of PDP11 memory. It should be the first word of an instruction.

Usage: This command will restart execution of the program at the specified location. If an illegal address is specified, a program interrupt will occur at that address .

Example(s): GOTO 200
GO +246

GR

Command Description

Purpose: To display the contents of the saved registers or a specified register.

Prototype: GR [{register}]

Parameters:

register is the number of a saved register which is to be displayed. It should be between 0 and 10, inclusive.

Usage: If a register number is specified, its contents will be displayed. If no parameter is given, all registers will be displayed. The register number 10 refers to the PS.

Example(s): GR 10
GENR

MODGR

Command Description

Purpose: To modify the contents of a saved register.

Prototype: MODGR {register} {value} [{value}]

Parameters:

register is the number of the first register which is to be modified. It should be between 0 and 10, inclusive.

value is the value that register is to be modified to.

Usage: The specified register(s) are modified to the given value(s). The register number 10 refers to the PS.

Example(s):
```
MODGR 0 2
MR 7 200 160
```

OSWIT

Command Description

Purpose: To perform a WARMSTART of the operating system.

Prototype: OSWIT

Usage: This instruction will restart OSWIT at the WARMSTRT address. The program, if loaded, is not destroyed.

Example(s):
```
OSWIT
OT
```

STEP

Command Description

Purpose: To step through execution of a program.

Prototype: STEP [{steps}]

Parameter:

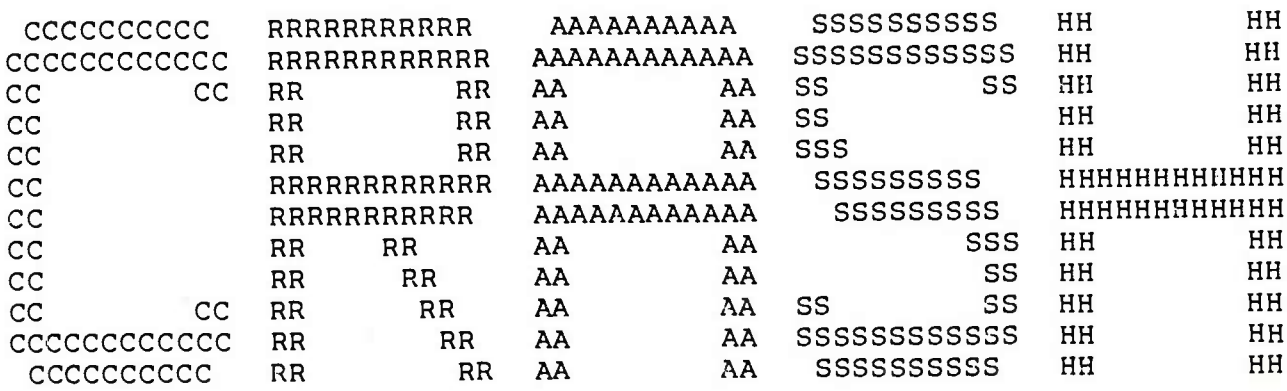
steps is the number of instructions to execute. If left out it defaults to 1.

Usage: This command will execute the specified number of instructions in the program. If no number is given, the next instruction in the program is executed. When the stepping is completed, debug is re-entered. If a breakpoint is encountered while stepping, it is recognized.

Example(s):
```
STEP 6
SP
SP 377
```

INDEX

APPENDIX B



Compiler for a Real time Applications SHop

U S E R S

M A N U A L

August, 1977

Richard A. Volz
Ralph E. Richardson

Revised: August, 1979

TABLE OF CONTENTS*

---

* Page numbers in the Table of Contents refer to the numbers in the lower corner of each page.

CHAPTER 1 - CRASH: AN OVERVIEW

Introduction

The field of digital computers and their application is perhaps the most dynamic field in engineering at the present time. Driving this change during the past five years has been the introduction and widespread acceptance of the microcomputer. Already there are numerous products on the market using microcomputers, and the future is almost limitless. At present, however, software support for these systems lags far behind their older and larger counterparts. Most software development for microcomputers is still done in assembly language. The availability of compilers for microcomputers is rather limited. Typical examples are the PL/M compiler and BASIC. These, however, are not really suited to most of the real time applications which are forthcoming for microcomputers.

With the broad range of applications to be developed for microcomputers during the next decade, it is important that the conveniences and documentation support of higher level languages be afforded the developer of microcomputer applications. The greater programmer efficiency of higher level languages, the dropping cost of memory, and the high cost of maintaining assembly language programs will almost certainly make it more cost effective to do most of the development in higher level languages.

CRASH (Compiler for a Real time Applications SHop) is a compiler being developed at Michigan to meet the needs of a higher level language for the Digital Equipment Corporation LSI-11 microcomputer. The compiler was designed and written by a CICE/CCS 675 class offered by Professor Richard Volz consisting of: Jack Bonn, Brian Cashman, Ted Kowalski, Alex Kushner, Steve Medlin, Bert Moberg, John Puttress, Al Segal, Jim Sterken, Dave Sun and Dave Yeager. The macro processor was written by a CICE/CCS 575 group consisting of Rick Richardson and Dave Smith. The compiler was revised, debugged, and maintained by Rick Richardson during 1977 and 1978, and by Terry Rosenbaum during summer, 1979.

The size of present microcomputers precludes the running of any significant software development system on the micro itself. Rather, the natural procedure is to use a cross compiler which runs on another machine. CRASH is a cross compiler which runs on the Michigan Terminal System (MTS) and produces LSI-11 assembly code. This is in turn run on the *11ASR assembler available under MTS. The object code produced by the assembler is link editted under MTS and either punched on paper tape (for loading on the LSI-11) or sent via a phone line directly to the

LSI-11.

1.1 Real Time Operations

According to Martin, a real time computer system is one which accepts inputs from one or more sources, acts upon these inputs, and produces corresponding outputs fast enough to effect the source. This definition encompasses a wide variety of systems. Between 2 a.m. and 4 a.m. (and upon a few other rare occasions) MTS may be considered a real time system. Other examples would include the use of a computer as a data concentrator, as the control element in a feedback loop, as a data logger for some real time process, or as a supervisor for a set of other real time computers.

There are two primary characteristics which distinguish real time applications from the scientific computations with which most programmers are familiar: the need to respond rapidly to the occurrence of asynchronous events external to the computer; and the need to handle I/O for a (potentially) large number of external devices in a manner which doesn't lock up the CPU during the I/O transfer. An example would be to require a computer controlling electric power distribution to suspend normal program operation upon the detection of a generator failure and initiate an orderly shutdown procedure for that generator and a redistribution of the load among the remaining generators. The consequences of these characteristics are far reaching.

First, in order to allow the user to specify the response to asynchronous external events, he/she must be given some control over interrupt handling. Secondly, since the computer is usually much faster than the devices it controls or responds to, it is common to have a single computer control several, or even hundreds of, external devices. As a result, one usually has several more or less independent pieces of code which are to run at different times. This leads to the concept of a task which is merely a named piece of code. In CRASH a task is a named procedure (program) which has no arguments. CRASH provides a mechanism for associating a task (i.e. a program) with an interrupt for a given external device. When an interrupt occurs, the program currently operating may be suspended and the associated task executed. When this task is completed, its execution is terminated and the original task resumed.

Associated with the notion of a task is that of a priority. If there are two or more tasks wishing to execute the CPU (the original and an interrupting task) there must be some mechanism for arbitrating which is to execute. In CRASH this is handled by having the user assign a priority to each task. Once started, a

Martin, James, Design of Real Time Computer Systems, Prentice-Hall

task will run to completion unless interrupted by a task with a higher priority. If task A has priority 10 and is interrupted by task B which has a priority of 25, task B will execute to completion unless interrupted by a task of priority greater than 25. When task B finishes, task A will be resumed.

Another use of tasks supported by CRASH is with synchronous timing. The LSI-11 microcomputer hardware for which CRASH is intended has a programmable real time clock. This allows the user to set a time interval in the clock and have the CPU receive an interrupt from the clock when that interval has past. CRASH utilizes this facility to allow the user to specify that a task is to be executed repeatedly (synchronously) at fixed intervals of time. Of course, the user must take care that the task can be completed before the next occurrence of that task.

Naturally, CRASH also provides a mechanism for cancelling a task if it is no longer needed.

The second area of importance was I/O. It is common for a real time computer to deal with a number of different devices. Each device may involve a sensor and an analog to digital converter, or a digital to analog converter and power amplifier. In either case there is likely to be a calibration offset and scale factor associated with the physical device which must be applied to every value in order to put it in the proper engineering units. Or it may be that each bit of the I/O word is associated with a different physical device (e.g. a switch).

For example, suppose one had a potentiometer mechanically connected to the output shaft of a rotational servo system. Electrically, the potentiometer might be connected between ground and 10 volts. An analog to digital converter connected to the potentiometer arm could be used to give a digital value for the position (angle) of the servo system. If the analog to digital converter were eight bits, 255 would correspond to 359 degrees. Obviously a scale factor of 360/256 must be applied if the converted number is to be viewed in degrees. Moreover, if the desired range is -180 to +180 degrees, an offset of -180 degrees may need to be applied.

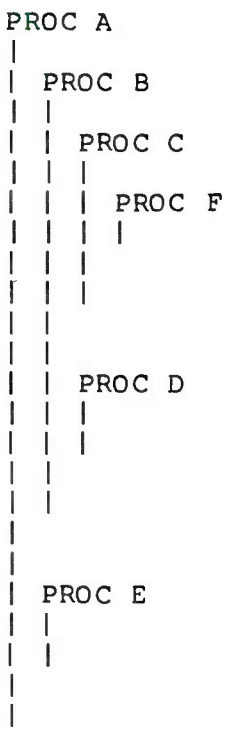
Another common requisite in the case of periodically sampled external signals is the use of past values of the signal as well as the current (most recent) sample. For example, suppose one is building a digital control system in which the control signal (digitally computed) depends upon the integral of the error (to force the long term behavior to have zero error) and the derivative of the error (to anticipate the changes which are going to occur and take the proper corrective action sooner). If the task which computes the control signal is executed periodically, past values of the error will clearly be required(to form the derivative).

CRASH has mechanisms to support these various I/O

requirements. There are variable attributes ANALOG and DISCRETE which allow a variable to be associated with a specific logical I/O unit at declaration. Moreover, a scale factor and offset may also be specified. After declaration, any I/O reference to the variable will automatically apply the offset and scale factor defined and use the given device. These characteristics need not be specified every time an I/O operation is done. In addition, a past history will automatically be kept for all analog variables. The history will automatically be updated every time an input operation is done on that variable. A convenient means of referencing past as well as current values is provided.

1.2 Language Structure

Crash is a block structured language similar in many ways to PL/I or XPL. A program consists of a list of procedure (subroutine) definitions, and is itself considered to be a procedure. Procedure definitions may be nested up to four levels. A procedure call is legal only to procedures defined immediately within, on the same or an outer level which does not enclose the procedure. For example, consider the situation shown below:



Procedure C may call procedures D, F, E, or B (recursive calls are allowed). Procedure B may call procedures C, D, or E, but not F. Procedure E may call B, but not C or D. In addition, any of the procedures in the above example may call a procedure defined and compiled external to procedure A.

Each procedure consists of a list of variable declarations followed by a list of statements. All variables used in a program must be declared. The purpose in this restriction is to try to enforce good program structure. Users are strongly encouraged to use the comment facilities of the language to insert descriptions of the use of all variables at the beginning of each procedure in the declaration section.

Every variable declared in a procedure is known without further declaration to all procedures defined within that procedure. For example, in the procedure structure given above, all variables declared in procedure A are known to procedures B, C, and F, unless explicitly declared within the latter procedures. Variables declared in procedure B are known to procedure C but not to procedure A. These are the usual rules of variable scope as per ALGOL, PL/I or XPL.

To allow common variables between externally compiled programs, a variable may be declared GLOBAL in two or more externally compiled procedures, in which case it will be known to all procedures in which it is so declared. There must be one procedure designated as the MAIN procedure in which all GLOBAL variables are declared. However, in other external procedures only those variables used need be declared.

1.3 Data Types

The basic data types of the language are REAL, INTEGER, BIT, BOOLEAN, and CHARACTER. A variety of subtypes may be obtained by including various attributes with the declaration of the variable. The legal attributes are:

ANALOG
DISCRETE
LDN
SCALE
OFFSET
MAP
DELAY
WORD
BYTE
CLAMP
PACKED

Some of the attributes require an argument, as indicated below.

The purpose of these attributes is to add flexibility to

the language. Indeed, many of the unique features of the language are achieved through use of these attributes. However, not all attributes may be used with all data types. A precise description of the rules of use are given in chapter 2. The purpose here is merely to give a rough idea of the intended use.

The ANALOG attribute may be used with either INTEGER or REAL variables. The DELAY attribute is used with it to specify a delay depth for the variable, i.e. the number of past values of the variable to be kept, as noted above. Clearly, DELAY is one of the attributes which must have an argument.

LDN, BYTE, WORD, SCALE, OFFSET, and CLAMP are used to associate a variable with a particular device. LDN specifies the logical device number. Every I/O reference to the variable will be through the given LDN. If desired it is possible to reset this under program control. The WORD and BYTE attributes specify the type of I/O which is to be used with the device. WORD specifies 16 bits, while BYTE refers to 8 bits of data. The SCALE and OFFSET are to provide compensation for any scale factor or offset effects of the particular sensors or analog to digital converters used. They will be automatically applied every time an I/O operation is done with the variable. This will alleviate the need of the user remembering to do this in a consistent manner throughout the program. CLAMP is used to restrict the output values to a device to a specified range. If a value is to be output which is outside this range, the highest (or lowest) clamp value is used.

The MAP attribute is normally used with DISCRETE variables, which must, in fact, be integers. This facility allows the user to assign a name to contiguous bits which make up the variable. The user may then refer to collections of bits of a variable by name in his program. This is particularly useful when dealing with digital I/O where each bit, or each small group of bits in a word may have independent meanings. For example, bits 0-7 may be a physical device number, bit 15 a state, and the other bits unused. The MAP facility would let the user name these individual components of the word.

The REAL, INTEGER, BIT and CHARACTER variables without any further attributes are roughly the same as in most languages. All REAL variables are 32 bits long. All character variables may be of length 0 to 255, but that length must be fixed at declaration time. BIT variables may have a maximum length of 16. All INTEGERS are of length 16 bits.

1.4 Arithmetic Processing

The usual arithmetic operations and order of precedence (e.g. as in FORTRAN) hold among INTEGER and REAL variables. Mixed mode expressions are allowed with conversions performed as required. The usual logical operations AND (&) , OR (|) and NOT (~) may be used with BOOLEAN variables. The relational operators =, >, <, >=, <= may be used and carry the usual meanings.

In addition to the normal operations, certain of the arithmetic and logic operators are extended to array operations. Addition (+), subtraction (-), AND (&), and OR (|) operations between arrays of matching dimensions are allowed. In performing these operations ANALOG variables are treated as one dimensional arrays of size specified by the depth of delay defined. Multiplication between a scalar and an array, and division of an array by a scalar on the right are defined.

Finally, multiplication between one and two dimensional arrays of appropriately matching dimensions is defined. Consider A*B. If A is a one dimensional array and B a matrix, then the result is treated as a row vector. If A and B are both vectors, then the inner product is formed. Finally, if A and B are both matrices of matching dimensions, the matrix product is formed. At present, array operations are handled by explicit subroutine calls (much faster than user written code). Eventually, they will become part of the language.

1.5 Control Structures

The language provides a variety of control structures to facilitate structured programming. It has the normal DO with iteration count of FORTRAN. This has been extended to the form DO I=I1,I2,...In, where the loop is executed for exactly those values of I contained in the list. In addition DO WHILE exp and DO UNTIL exp constructs have been added, where exp is a logical expression which controls when the loop processing ends.

A DO CASE exp is provided. Exp is an integer valued expression. A series of statements follows the DO CASE. Exp is used to select exactly one of these for execution. Since each of these statements may in fact be a nested group of statements, great flexibility is provided.

Two forms of the IF statement are provided: IF exp THEN s1; and IF exp THEN s2 ELSE s3. Exp is a logical expression taking the values true or false. s1, s2 and s3 are statements whose execution is controlled by the value of exp.

1.6 Procedures And Tasks

The language supports two types of transfer of control to other sections of code. The first is via a procedure call (subroutine call) and is accomplished in the same manner as in FORTRAN. The procedures may be subroutines or functions and the use within a statement dictates which is expected. As discussed above the procedures may be either internally defined or externally defined.

The second mechanism for transfer of control is in response to an external event or internal processing condition (e.g. array reference out of subscript range). Most often this uses a TASK. A TASK is nothing more than an externally defined procedure which takes no arguments. To be established the name of the TASK (procedure) is used in one of several SCHEDULE statements. The SCHEDULE statements state the conditions (e.g. interrupt on device number 3, or every .25 seconds) under which the background program is to be interrupted and the TASK named started.

Since the potential exists for several conditions to be satisfied at nearly the same time, resulting in more than one task trying to execute at the same time, each SCHEDULE statement includes a specification of a PRIORITY of that task. The operating system which runs with the compiled code will ensure that the highest priority task will be the one that executes. It will execute to completion unless interrupted by a higher priority TASK (i.e., there is no time slicing).

1.7 Input/Output Considerations

There are two basic types of I/O supported by the language. The first assumes that the variables used in the statement have had an LDN defined for them(a default of the console device is assumed). This type takes the form GET <variable list> or PUT <variable list>. All variables in the list are processed to the LDN given in their declaration (unless reset) and the OFFSET and SCALE applied. This is the form one would normally use in communicating with external devices other than the control console.

The second I/O structure is intended to be used primarily with the console device. The forms are <variable> = INPUT; and OUTPUT = <expression>. On INPUT, the values are assumed to be in character form separated by either comma or blank delimiters (stream I/O in PL/I terminology). On OUTPUT all variables are converted to character form and automatic spacing applied. This relieves the user of worrying about complicated format structures. The default device for INPUT/OUTPUT is the console device.

1.8 CRASH Programming Conventions

With some programming languages, like FORTRAN, the programmer has a fairly rigid format he must follow when writing his program. In CRASH, the source code is considered as a continuous stream of characters. Card boundaries are completely ignored. It is therefore possible to have several statements on one line, or none at all. Blanks may appear anywhere in the source, and in any number. For example, the following three statements are identical to CRASH:

```
ALPHA(I) = BETA(J);

ALPHA(I)=BETA(J);

ALPHA     (     I)     =BETA(     J      )      ;
```

Clearly, the first example is the easier to read. However all three are legal and produce the same results.

The problems with this type of parsing are that, left to themselves, different programmmers will come up with different formats for their programs. Some conventions have been set down for using structured languages like CRASH, and the programmer should try to stick to these, or at least be consistent so that his programs can be read by someone else.

Comments can be freely used wherever needed, and should be used generously. They should be not only a guide to the reader, but a guide to the programmer as well. They should describe the process being performed, and not simply restate what a particular statement does. A comment is started by a /*. All text after the /* is ignored by the compiler except for control toggles (chapter 9). The comment is ended by the first */. Comments can be as long as desired and will be printed on the listing of the program.

CHAPTER 2 - DATA TYPES AND STRUCTURES

Introduction

This chapter covers constants, identifiers, variable types, storage allocation, declaration statements, and reserved words.

2.1 Constants

General Description

Constants are quantities whose values are known and invariant. CRASH supports four types of constants.

Precise Description

A. Integer Constants

An integer constant is a stream of decimal digits (0-9) preceded by an optional sign (+ or -). Internally, integer constants are stored in 16-bit two's complement form. The range of allowable values is -32768 to +32767. Blanks, commas, and decimal points are not allowed within integer constants.

Examples:

```
212
-32768
+10
```

B. Real Constants

A real constant is a stream of decimal digits (0-9) and a decimal point (.) followed by another stream of decimal digits. The constant may optionally be preceded by a sign (+ or -). A power of ten exponent may be specified by immediately following the constant with "E", an optional sign (+ or -), and a one or two digit exponent. Internally, real constants are stored in LSI-11 floating point form. The range of values expressible is approximately 10**(-38) to 10**(38). Blanks are not allowed within real constants.

Examples:

```
.0073
-93.
+6.8
1.E35
5.7E-6
```

C. Bit Constants

Bit constants provide alternative notations for 16-bit constants. They consist of the letter B, O or H followed by a stream of digits enclosed in quotes ("). Blanks may be used freely between the quotes for clarity. If the first character is a B then the digits are assumed to be binary (0,1), else if the first character is an O they are assumed to be octal (0,1,...,7), and if the first character is an H they are assumed to be hexadecimal digits (0-9,A-F). If no letter appears, an O is assumed by default.

If more than 16 bits are specified, the rightmost 16 bits are used; if less are specified, they are stored right justified in a 16-bit field with the unspecified bits zeroed out.

Examples:

```
"17077"
B"0001 1110 0011 1111"
H"1E3F"
O"17077"
```

D. String Constants

A string constant is a stream of characters enclosed in primes ('). Strings of length 0-255 are allowed, a length of zero denotes the null string. Two primes in a row ('') within a string may be used to denote the occurrence of the single character prime. Since the omission of the closing prime is a common programmer error, the string is automatically terminated at a single semicolon to aid error recovery. Therefore, semicolons must be doubled if they are to appear in a string constant.

Examples:

```
'ABC'
''
'01+173'
'AIN''T NO MORE'
```

2.2 Identifiers

General Description

Identifiers are used in CRASH to represent variables, labels, procedures, tasks, and macros.

Precise Description

Identifiers occur in CRASH as names of variables, procedures, tasks and macros. For a complete description of the use of macros, see Chapter 10. An identifier is a stream of from 1 to 255 alphabetic (A,...,Z,_,$) or numeric (0,...,9) characters the first of which must be A-Z or $. When an identifier is used to name an external procedure, task, or GLOBAL variable, it is restricted to a maximum length of 8 characters.

Examples:

```
CENTS2$
START_UP
A
WHO_NEEDS_AN_IDENTIFIER_THIS_LONG
```

2.3 Variable Types And Storage

General Description

A variable is a quantity which can take on different values throughout the execution of a program. An identifier is used as the name of a variable, which ultimately refers to a location in memory where the value of that variable is stored.

All variables must be declared before their use via declaration statements (see section 2.4). The amount of storage allocated as well as the internal format used is determined by the "type" of the variable and its "attributes". This information is included as part of the variable's declaration. This section discusses types, attributes, and their combinations. The actual syntax for declarations, however, is not discussed until section 2.6.

Precise Description

A. INTEGER Variables

Integer variables refer to 16 bits (2 bytes) of storage. They are stored in two's-complement format internally. The range of values that can be represented is therefore -32768 to 32767. Overflow of an integer variable is not detected at run-time, it is the programmers responsibility to determine the range of values needed and to chose the appropriate type (INTEGER or REAL) which will allow arithmetic without overflow. Note that whenever possible INTEGER should be used in preference to REAL since arithmetic operations on integers are much faster on the LSI-11 than floating point operations on real numbers.

B. REAL Variables

Real variables refer to 32 bits (4 bytes) of storage. They are stored in LSI-11 floating-point format.

C. CHARACTER Variables

Character variables refer to up to 255 bytes of storage depending on the maximum length specified in the declaration statement. The number of bytes allocated is equal to the maximum length specified (rounded up to the nearest even number actually) plus two additional bytes for holding the maximum length and the current length of the string. The storage allocated for a character variable (string) is fixed by the declaration, not by the current length of the string. All character variables are initialized with a current length of zero unless an INITIAL value is specified (see section 2.6.B.5).

Note: Due to implementation restrictions CHARACTER ARRAYS may only have a maximum length of 254 bytes.

D. BIT Variables

Bit variables refer to 16 bits (2 bytes) of storage. There is a length specification in declarations for bit variables, but this has no effect on the amount of storage allocated; values are always stored right-justified in bit variables. Bits are numbered from 0 to 15, with bit 0 being the least significant bit, and bit 15 the most significant (BEWARE: Reverse IBM notation).

E. ANALOG and DISCRETE Variables

The ANALOG attribute is used together with certain other attributes (described in section 2.6B) to associate additional information with a REAL or INTEGER variable. Analog variables are typically associated with an I/O device such as an A/D or D/A converter. In particular, ANALOG variables allow the particular characteristics of the device to be compensated for automatically during an I/O operation.

DISCRETE variables are similar to ANALOG variables in that they are associated with a particular device, however they can be INTEGER only. They are intended to be used with status settings of a device.

Both ANALOG and DISCRETE variables belong to a class of variables known as DELAY variables. Using the DELAY attribute (described below), a circular list structure of past values (history) of the variable can be kept.

See the Microcomputer Handbook, Digital Equipment Corp. 1977

2.4 Variable Organization

General Description

A CRASH variable may be either a scalar in which case it has a single value or an array in which case it has a collection of values.

Precise Description

A. Scalars

Scalars refer to only a single value. The size and internal storage layout is determined by the scalar's type and attributes.

Examples:

```
SCAL, I, J, COUNT
```

B. Arrays

With arrays, a single variable name is used to refer to more than one element of storage. The number of elements is determined by the dimensions of the array. Each dimension consists of a lower (optional) and upper bound for subscripts to be used in that dimension. The default lower bound is 0.
For example, the array named VEC, with one dimension consisting of a lower bound of -1 and an upper bound of 3, refers to five elements of storage, namely VEC(-1), VEC(0), VEC(1), VEC(2), and VEC(3).

Internally, arrays are stored in row major form, i.e. with the value of the last index increasing most rapidly. For example, the array named A with three dimensions each having a lower bound of 1 and an upper bound of 2 appears at ascending storage locations in the following order:

```
A(1,1,1), A(1,1,2), A(1,2,1), A(1,2,2),
A(2,1,1), A(2,1,2), A(2,2,1), A(2,2,2)
```

An array specification is made by including a list of dimensions as an attribute in the declaration statement.

C. Delay Variables

Delay variables allow the past values (history) of a variable to be maintained in a circular list structure. Associated with the variable are several elements comprising the history, a delay size limiting the number of values in the history, and a current element pointer. Whenever the variable is referenced by name without a delay specification the current element is used. Assignments and input to a delay variable cause a new current element to be computed.

Internally, delay variables occupy a block of storage big enough to contain the maximum number of previous values to be kept. The current element will initially point to the first element in the block of memory, unless INITIAL values were specified. In the case of INITIAL values, they will occupy the first elements of the list with the current element pointer pointing to the last initial element (the first one specified in the declaration). When a new current element is needed, the current element pointer is incremented by the size needed by a variable element (1 or 2 words). If the pointer is now beyond the end of the block of memory assigned to the variable, the size of the delay variable storage area is subtracted from the current element pointer, wraping it back around to the first element in the list. In this way a circular list is formed containing the current and previous values of the variable. Any element in the list may be referenced relative to the current element by using the delay specification (see chapter 8).

2.5 Storage Allocation

General Description

The way in which the storage area for each variable is allocated can be controlled by use of the AUTOMATIC, STATIC, and GLOBAL attributes. If none of these three is specified for a variable, AUTOMATIC is assumed. One exception to this rule is that in MAIN procedures, STATIC is the default and is used even if AUTOMATIC is specified.

Precise Description

A. Automatic Storage Allocation

Storage for automatic variables is allocated (and initialized if an INITIAL attribute was provided) whenever the procedure in which the variable was declared is activated (i.e. called as a subroutine or function or invoked as a task), and freed for reuse by other automatic variables in other procedures or tasks when the procedure is deactivated (i.e. when it executes a RETURN statement).

The advantage of AUTOMATIC variables is that the same space can be used at different times by many automatic variables. The disadvantages are that variable values are not preserved from one call (or task invocation) to the next and that there is more overhead associated with setting up AUTOMATIC variables each time a procedure is entered.

The AUTOMATIC storage mode is particularly useful with large arrays which are being used within a procedure for intermediate calculations and which need not keep their values from call to call. Another application might, for example, be a situation where several tasks with large individual storage requirements are to be scheduled to execute in succession every once in a

while. If AUTOMATIC storage mode is used for most of the variables, execution can proceed without problems even though memory might not be large enough to hold all of the task storage simultaneously.

B. Static Storage Allocation

Storage for STATIC variables is allocated when the program is loaded; the storage then remains allocated until the program is unloaded. STATIC variables are initialized (if an INITIAL attribute was provided) only once at load time. STATIC variables should be used in preference to AUTOMATIC variables whenever possible since they are referenced more efficiently, occupy less space, and require less overhead for initialization.

C. Global Storage Allocation

Storage for GLOBAL variables, like STATIC variables, is allocated when the program is loaded, and remains allocated until the program is unloaded. GLOBAL variables behave exactly like STATIC variables except that they can also be referenced, outside of the normal rules of scope, by any other task or procedure, internal or external, in which they are declared. For example, if variable X is declared to be GLOBAL in two external procedures, both procedures will be referencing the same location in storage when they use X while if X had been declared to be STATIC there would be a separate version of X for each procedure. GLOBAL variables, along with parameters, are the only way that tasks and external procedures can communicate with each other.

All GLOBAL variables used in a program must be declared and initialized if desired in the MAIN procedure. Each declaration for the same GLOBAL variable must have the same type and attributes (be careful to match these declarations properly, CRASH does not check this since each task or external procedure is compiled separately). GLOBAL variables names must be 8 or less characters in length.

2.6 Declaration Statements

General Description

Declaration statements are the means of associating types and attributes with identifiers for variables used within a procedure or task. Declaration statements are required at the beginning of every CRASH procedure or task for all variables which are intended to be defined within the procedure. All variables must be declared before they are used.

The basic format of a declaration statement is:

```
<type>  <identifiers>  <attributes>,
        <identifiers>  <attributes>, ... ;
```

Where:

<type> specifies the type of all variables declared in this statement (up to the semicolon).

<identifiers> is either a single identifier or a list of identifiers separated by commas and enclosed in parentheses.

<attributes> specifies a list of attributes, separated by blanks, to be associated with the preceding <identifiers>.

Thus, declaration statements may look like:

```
type identifier ;
type (identifier, ..., identifier) attribute ;
type identifier attribute...attribute, identifier attribute;
type (identifier, identifier), identifier attribute ;
type identifier, identifier, identifier ;
```

The following two sections discuss the details of specifying <type>s and <attributes> in declaration statements to describe the variable types discussed in section 2.4.

Precise Description

A. Types

The first word in a declaration statement specifies the type for the entire statement (up to the semicolon). Valid specifications for types are:

```
INTEGER
REAL
BIT(i)          1<= i <=16     i=maximum no. of bits
CHARACTER(n)    0<= n <=255    n=maximum character length
ROUTINE
TASK
```

The first four of these tell the CRASH compiler what element size is to be used as well as in what internal format the data will be stored for each identifier declared in this statement. For instance, INTEGER specifies an element size of 2 bytes and a two's complement internal format whereas REAL specifies 4 bytes and LSI-11 floating-point format. BIT(i) always denotes an element size of 2 bytes no matter what "i" is.

For scalars, the element size is the size of the variable; for arrays the element size is the size of a single element (A(I,J)) of the array. For procedures, the element size is the size of the value which the procedure returns if it is called as a function. For delay variables, the element size is the size of a single "history" (delay) element.

"i" and "n" are both integer constants. If i>16 or i<1, a

warning message is printed and 16 is assumed. If n>255 or n<0, a warning message is also printed and 255 is assumed. When declaring procedure parameters, "i" and "n" should be replaced by asterisks (*) (which signal that that value has already been specified) since the lengths declared in the calling program program will be used. If numbers are used instead of asterisks, a warning message will be printed and the numbers will be ignored.

ROUTINE is used to declare procedures which do not return values. Likewise, TASK is used to declare tasks which, presumably, will be referenced in scheduling statements, and which do not return values. In both cases, an element size is not needed. Procedures which return values are declared by specifying the type (REAL, INTEGER, or CHARACTER) of value to be returned, and additionally specifying either the INTERNAL or EXTERNAL attribute (see chapter 3).

Examples:

```
CHARACTER(26) ALPHABET;
INTEGER (I,J,K);
BIT(8) MASK,HEX_CON;
ROUTINE (PROC1,PROC2);
BIT(*) PARM;
```

B. Attributes

For each group of one or more identifiers within a declaration statement, a list of attributes can be specified. If more than one attribute is specified, the attributes should be separated by blanks. The list is terminated by a comma or the end of the statement (;). Attributes can appear in any order except for 4 cases mentioned below. Attributes specified more than once will cause a warning message to be printed.

B.1. ANALOG, DISCRETE, LDN, DELAY, SCALE, OFFSET and CLAMP

Analog variables are declared by specifying the ANALOG attribute with an INTEGER or REAL type. A fully specified declaration for an analog variable, ANGLE, would appear as below:

```
type ANGLE ANALOG DELAY(d)
        SCALE(s) OFFSET(o) LDN(n) CLAMP(m,p) iotype;
```

Where:

"type" is either INTEGER or REAL

"s" is a real constant specifying the scale factor. If the SCALE attribute is omitted, "s" defaults to 1.0.

"o" is a real constant specifying an offset value. If the OFFSET attribute is omitted, "o" defaults to 0.0.

"d" is a positive integer constant specifying a delay value. If the DELAY attribute is omitted, "d" defaults to 1.

"n" is an integer constant between 0 and 30 specifying a

logical device number. If the LDN attribute is omitted, "n" defaults to -1 which will remain unassigned until runtime. If I/O is attempted with an LDN of -1, an error will be recognized by the operating system, and program execution will terminate.

"m" is an integer constant specifying the lowest value to be output to a device.

"p" is an integer constant specifying the highest value to be output to a device. If the CLAMP attribute is omitted, "m","p" default to -32768,32767 (no clamping).

"iotype" is either BYTE or WORD. If omitted, the default is WORD. See section B.7 below.

The SCALE, OFFSET, BYTE, WORD, and CLAMP attributes are used only during I/O. More information about their use appears in chapter 7.

If the analog variable being declared is a procedure parameter, the LDN, DELAY, SCALE, OFFSET, BYTE, WORD, and CLAMP attributes should not be specified since they will take their values from the declaration in the calling program. These attributes may be specified in any order.

Discrete variables are declared by specifying the DISCRETE attribute with an INTEGER type. A fully specified declaration for a discrete variable, STAT, would look like:

```
INTEGER STAT DISCRETE DELAY(d) LDN(n) iotype;
```

Where the rules for "d", "n", and "iotype" are the same as with ANALOG declarations. These attributes may appear in any order. DISCRETE variables are always INTEGERS.

Examples:

```
INTEGER COFFEE_TEMP DISCRETE BYTE LDN(1);
INTEGER ANTENNA_POSITION ANALOG DELAY(10)
            WORD SCALE(.6) OFFSET(180.) LDN(12);
```

B.2. GLOBAL and STATIC Attributes

These attributes can be used to specify a GLOBAL or STATIC storage allocation scheme. These attributes are valid only with INTEGER, REAL, BIT, and CHARACTER data types and are illegal with types TASK and ROUTINE.

Examples:

```
REAL (SW1, SW2) GLOBAL;
BIT(8) MASK STATIC;
```

B.3. INTERNAL and EXTERNAL Attributes

The INTERNAL and EXTERNAL attributes are used to declare procedures. The type, in this case, refers to the type of the procedure's returned value (if any). The type ROUTINE should be used if the procedure doesn't return a value. In procedure declarations, one of these two attributes must appear unless the type is ROUTINE or TASK; in that case, if no attribute is specified, the default is EXTERNAL. Note that if type TASK is specified, INTERNAL will be an illegal attribute since tasks are always EXTERNAL. The INTERNAL or EXTERNAL attribute, if present, must be the first attribute after the identifiers in the declaration statement. This constraint is reasonable since no other attributes are legal in a procedure declaration. An INTERNAL procedure is one which is defined inside the scope of the procedure which is declaring it. EXTERNAL procedures are defined outside the scope of the declaring procedure and may be compiled separately from the declaring procedure.

Examples:

```
REAL INT_TO_REAL INTERNAL;
INTEGER REAL_TO_INTEGER INTERNAL;
ROUTINE (INIT, CLOS);
ROUTINE UPDATE INTERNAL;
TASK CONTROLLER EXTERNAL;
TASK (READER, WRITER);
```

B.4. Array (Dimension List) Attribute

Arrays are declared by specifying a dimension list as an attribute in the declaration statement. The syntax for a dimension list is as follows:

```
(dim, dim, dim, ...)
```

Where:
"dim" denotes either "low:high" or just "high"
"low" is the lower bound
"high" is the upper bound

"low" and "high" are either integer constants with an optional sign or unsigned integer variable names. If "low" is not specified, 0 is assumed. Up to 62 dimensions are allowed.

As was the case with the CHARACTER and BIT variable types, arrays which are procedure parameters must be declared with asterisks in place of the usual "dim" groups since the dimensions declared in the calling program are the ones that will be used. If asterisks are not specified for parameter arrays, a warning message will be printed.

The dimension list attribute must appear as the first attribute after the identifiers in the declaration statement. Note also

that this attribute is meaningless with the ROUTINE or TASK type since procedures can only return scalar values.

Some special comments are in order about the use of integer variable names within an array dimension list. This construct implicitly invokes an extended version of the automatic storage allocation scheme. With a regular automatic array (with constant dimension bounds), the size of the array, but not the storage location, is known at CRASH compile time. But when integer variables are present in the dimension list (dynamic arrays), neither the size or the location is known. Naturally, when fewer things are known at compile time, more things must be done at run time resulting in less efficient code with more overhead. Thus, dynamic arrays should only be used when storage is very tight since they do tie up less space with large arrays especially. (For example, if a number N which might vary from 50 to 500 is to be read in, and a vector with N elements is required to handle the corresponding data, it would be worthwhile to make that vector dynamic with an upper bound of N rather than to just make the upper bound 500 and always reserve 500 elements for the worst case.) Due to the way dynamic arrays are implemented, dynamic arrays must always be automatic, and the integer variable dimensions used must always be previously initialized GLOBAL or STATIC variables, or automatic variables from lower level procedures whose values can be accessed through the usual rules of scope.

Examples:

```
REAL (INDEX,BRAND,PRICE,QUANTITY) (1:100) GLOBAL;
INTEGER YEAR (1976:1984) STATIC;
INTEGER AREA(1:DIM1,-1:DIM2,1:DIM3);
BIT(*) PARM (*,*,*);
```

B.5. INITIAL Attribute

Variables can be initialized to prescribed values via the INITIAL attribute, the syntax is:

```
INITIAL(cons,cons,cons,...)
```

Where:

"cons" denotes a signed integer or real constant or a string constant. A "cons" may be prefixed by "integer #" to specify a duplication factor on "cons". This makes it much easier to initialize a sparse matrix, or to initialize an array to a single value. The type must agree with the type of the declaration, however real-to-integer and integer-to-real conversions will be performed if necessary (with a warning message). If a string constant's length exceeds the maximum length declared, the string constant is truncated and a warning message produced. The INITIAL attribute must be the last attribute specified.

For scalars, only one "cons" is allowed. For arrays, the number of "cons" must not exceed the number of elements in the array; the constants are assigned to increasing array element storage locations starting at the first location (array elements are stored in row major form). There is no way to initialize selected individual array elements out of this order. For delay variables, the number of "con"s must not exceed the delay size as specified in the DELAY attribute. Initial values for DELAY variables are assigned from newest to oldest beginning with the current element.

The INITIAL attribute is illegal for:
---procedures
---dynamic arrays
---parameters
---GLOBAL variables, unless in the MAIN procedure

Examples:

```
INTEGER CONS10 STATIC INITIAL(10),COUNTER INITIAL(0);
CHARACTER(5) CMDS(3) INITIAL('BEGIN','END','HELP');
REAL (DIRECTION, TEMPERATURE, HUMIDITY) ANALOG
          DELAY(3) INITIAL (0.,0.,0.);
INTEGER MASK INITIAL(H"01FF");
INTEGER BIG(150) INITIAL(50#1, 50#2, 50#3, 0);
```

B.6. MAP Attribute

The MAP Attribute provides a way to name contiguous fields (Bit-strings, or character substrings) of an integer or character variable. There can be a maximum of 16 fields specified for each variable. These fields may be of any positive length, and may overlap each other. The CRASH scanner will generate a macro definition for each field description. This enables the user to refer to the specific bits or characters with a simple identifier, instead of having to put the field description along with every reference of the field. The syntax for a MAP attribute is as follows:

```
MAP(name[field], name[field], ... , name[field])
```

Where:
"name" denotes the name assigned to the substring of the variable being declared.
"field" denotes either:
start,length or start
"start" is the number of:
1) The first character in the substring.
2) The least significant bit in the bit-string
"length" is the number of:
1) characters in the substring.
2) bits in the bit-string.

Note: Characters are numbered starting with "0" being the first

character in a string. Bits are numbered with the least significant bit as zero and the most significant bit being fifteen. If no length field is specified, the field is taken as the rest of the string, or all bits from start to bit 15. It is also important that only one variable be declared per declaration statement when the MAP attribute is used, since there can be only one macro with "name".

MAP variables appear to be simple variables to the programmer, however there are some restrictions upon the use of MAP variables which do not apply to a simple variable. MAP variables may not be used to specify a parameter to a procedure (they may be arguments to a procedure, though), nor can they be used in a delay specification. They cannot specify a priority for a task or be used as a task identifier (see chapter 6). Except for these case, MAP variables behave as if they were simple variables.

Examples:

```
BIT(16) TRAIN MAP(SPEED[0,5],TRACK[6,6],DIR[5,1],BANK[12,4]);
CHARACTER(80) INBUF MAP(COMMAND[0,5],COMMENT[5]);
INTEGER TWO_BYTES MAP(LOW[0,8],HIGH[8]);
```

B.7. BYTE and WORD Attributes

The BYTE and WORD attributes are used with DISCRETE and ANALOG variables to specify the size of the data expected to and from the device connected to the LDN. The default is WORD (16 bits).

Examples:

```
REAL SERVO_POSITION ANALOG BYTE LDN(0);
```

2.7 Reserved Words

General Description

The following words are reserved by CRASH and cannot be used as variable, task, procedure, or macro names:

$COPY	AFTER	ANALOG	AND	AT
BIT	BY	BYTE	CALL	CANCEL
CASE	CHARACTER	CHECK	CLAMP	DELAY
DISCRETE	DO	ELSE	END	EVERY
EXIT	EXTERNAL	FOR	GET	GLOBAL
GO	GOTO	IF	IGNORE	IN
INITIAL	INPUT	INTEGER	INTERNAL	LDN
LENGTH	LOCK	MAIN	MAP	MAXLEN
MIN	MOD	MSEC	NEXT	NOT
NUMARGS	OFFSET	ON	OR	OUTPUT
PACKED	PC	PRIO	PROCEDURE	PUT
REAL	RECORD	RETURN	REVERT	ROUTINE
R1	R2	R3	R4	R5
R6	R7	SCALE	SEC	SP
START	STATIC	STOP	TASK	THEN
TO	UNLOCK	UNTIL	WAIT	WHILE
WORD	XOR	CARD		

CHAPTER 3 - PROCEDURES

Introduction

A CRASH program consists of one or more EXTERNAL procedures or tasks each of which may contain one or one more INTERNAL procedures. The INTERNAL procedures may themselves contain other INTERNAL procedures. Any procedure can be used as either a subroutine or a function and can, optionally, be given a list of arguments when it is invoked.

3.1 Procedure Types

General Description

Four types of procedures can be used: EXTERNAL, INTERNAL, TASK, and MAIN.

Precise Description

A. EXTERNAL

An EXTERNAL procedure is the basic unit processed by CRASH. A separate object module (an *11ASR assembly) is produced for every EXTERNAL procedure. These object modules need not be produced in the same CRASH run; several files containing separate object modules can be combined later using *LINK11.

B. INTERNAL

An INTERNAL procedure is a procedure which is defined within either an EXTERNAL or another INTERNAL procedure. A maximum nesting of 3 INTERNAL procedures inside an EXTERNAL procedure is allowed. Note that this constraint applies to nesting of procedure definitions only. An arbitrary nesting of procedure calls is allowed. The procedure definition nesting limitation is due to implementation restrictions and not imposed by the nature of CRASH.

C. TASK

A TASK is an EXTERNAL procedure which is intended to be set up by scheduling statements to execute independently at specified time intervals or as a result of an external event. Internally, a TASK looks the same as a regular EXTERNAL procedure except that it may not have any parameters passed to it upon invocation.

The PDP-11 Assembler and Link-Editor: A User's Guide to *11ASR and *LINK11, University of Michigan Computing Center Memo 286, 1973

Ibid.

D. MAIN

A MAIN procedure is also a type of EXTERNAL procedure. There must be one (and only one) MAIN procedure in every program. Program execution will begin with the first statement in the MAIN program. Internally, a MAIN procedure looks the same as a regular EXTERNAL procedure except that it has a special section containing GLOBAL variables associated with it. All global variables must be declared in the MAIN procedure, and if they have initial values they must be specified here. Variables declared in the MAIN procedure always have the STATIC attribute regardless of how they are declared since no advantage is gained in having automatic storage allocation for the MAIN procedure.

When the $D (debug mode) toggle is turned on, the code emitted for the MAIN procedure is modified so that upon entry to the MAIN procedure the DEBUG command system is invoked. Any procedures which are to be debugged using the CRASH DEBUG package must be declared in the MAIN procedure and must also be compiled with the $D toggle enabled. See chapter 9 for information of the use of control toggles and chapter 14 for a complete description on the use of the debugger.

3.2 Subroutines And Functions

General Description

While a MAIN procedure is invoked by the OSWIT RUN command and a TASK is invoked via scheduling statements, INTERNAL and EXTERNAL procedures are referenced as either subroutines or functions.

Precise Description

A. Subroutines

A procedure is referenced as a subroutine via a CALL statement. For example:

```
CALL PRC(A, 1., 2*B+C);
```

Where presumably the number and types of the arguments correspond to the parameters defined in the definition of procedure PRC. Subroutines generally pass back information by changing the values of some of the arguments or by setting commonly accessed GLOBAL variables. Arguments are passed by reference except for constants and expressions which are passed by value.

Operating System WIth Trains. See the OSWIT Users Manual.

B. Functions

A procedure is invoked as a function by the appearance of the procedure name (and associated arguments) in an expression. The procedure is expected, in this situation, to RETURN a scalar value having the type specified in the procedures's declaration. For example:

```
INTEGER PROC EXTERNAL;
...
VAL = 2 * PROC(A,B,7.);
```

Here procedure PROC is given 3 arguments and is expected to return an integer value which is multiplied by 2 and used as a new value for variable VAL. A procedure returns a value with a RETURN statement which is of the form:

```
RETURN expression;
```

The expression will be converted, if necessary to the type specified in the procedure definition statement. CHARACTER expressions cannot be automatically converted to INTEGER or REAL. The subroutines D2FLOAT and D2BIN (chapter 11) can be used to perform these conversions. Numeric to character conversions will be automatically applied. If there is no expression (i.e. "RETURN ;") the returned value is undefined. Usually this type of RETURN statement is used if the procedure is always expected to be called as a subroutine.

Note: the same procedure can always be called as either a function or a subroutine. If it is called as a subroutine and RETURNs a value, that value will be ignored. Functions can, if desired, change the value of parameters in the same way that subroutines do.

3.3 Procedure Definitions

General Description

A procedure is a sequence of statements headed by a procedure definition statement, followed (optionally) by declaration statements, followed by at least one executable statement, and terminated by an END statement. An entire procedure is itself treated by CRASH as an executable statement; thus, one of the executable statements within a procedure can be another procedure. This nesting of procedures is allowed to a maximum of four levels (the external procedures and tasks must always be at level 0, internal procedures can occur at levels 1, 2, and 3).

Precise Description

A. Procedure Definition Statement

A procedure definition statement has the following form:

```
label: type PROCEDURE (par1,par2,...,parn) [MAIN or TASK] ;
```

Where:

"label" is the name of the procedure. If the procedure is EXTERNAL, MAIN, or a TASK, the name must be 8 characters or less long and must be unique from the names of all other such names or GLOBAL variable names within the program.

"type" specifies the type of the scalar value to be returned by the procedure (i.e. INTEGER, REAL, BIT, or CHARACTER). If "type" is missing, it defaults to INTEGER.

"(par1, par2, ..., parn)" is the procedure's (optional) parameter list. "par1" ... "parn" are the names of parameters which will be passed to the procedure as arguments whenever it is invoked as either a function or a subroutine. Each parameter defined in this list must be declared in the declaration section of the procedure. The "pars" must be the simple name of a variable. Subscripted, sub-unit selected, and MAP variables may not be used to specify the parameter list of a procedure. If on a given invocation of the procedure, fewer arguments are provided than are defined here, the remaining parameters are left undefined. (the pre-defined function NUMARGS can be used to find the actual number of arguments that were passed.) No check is made to see if the type of each argument matches the type of its corresponding parameter as declared within the procedure; care must be taken to insure that these always match. Procedure names or statement labels cannot be passed as parameters.

If MAIN is present for EXTERNAL, level 0, procedures only, the procedure will be invoked as the MAIN procedure when the program is run. Similarly, TASK is used to designate an EXTERNAL, level 0, procedure as a task. MAIN procedures and TASKs cannot have parameter lists.

B. Declaration and Executable Statements

Declaration statements (if any) for parameters and local variables which are to be used by the procedure should come first. All parameters must be declared; no storage type (i.e. AUTOMATIC, STATIC, or GLOBAL) should be specified for parameters since the storage type of a parameter depends on the type of the corresponding argument, and could vary from call to call.

There must always be at least one executable statement in every procedure.

C. END Statem nt

ND statemen is of the form:

E D label ;

r :
" abel" is th name of the procedure as given earlier in the
c dure defi ition statement. This statement ends the
c dure. Note, while this statement is always the last line in
h procedure, it is not necessarily the last statement
c ted. A RE URN statement (there can be several in a
c dure) cause the procedure to return immediately to the
i ng program If execution flows into the END statement, a
T RN ;" is ma e.

3.4 Sample Program

```
TEST: PROCEDURE MAIN;          /* HAVE TO HAVE A MAIN PROCEDURE */
   REAL EXT_PRC EXTERNAL;      /* COULD BE COMPILED EARLIER     */
   INTEGER INT_PRC INTERNAL;   /* THIS IS DEFINED BELOW...      */
   ROUTINE EXT_SUB EXTERNAL;   /* JUST AN EXTERNAL SUBROUTINE */
   INTEGER (A,B,C,D);          /* DEFINE SOME LOCAL VARIABLES   */

   ....                        /* LEAVING OUT THE DETAILS ...   */

   CALL INT_PRC(A,B);         /* USE INT_PRC AS A SUBROUTINE   */
   C = INT_PRC(1,A+B/2);      /* USE INT_PRC AS A FUNCTION     */
   D = EXT_PRC(A,B);          /* INVOKE THE EXTERNAL PROCEDURE */
   CALL EXT_SUB(A,B);          /* CALL THE SUBROUTINE */

   ....                        /* ... MORE DETAILS ... */

   INT_PRC: INTEGER PROCEDURE(X,Y); /* PROCEDURE INT_PRC */
      INTEGER (X,Y);          /* DECLARE THE PARAMETERS   */
      INTEGER Z;              /* DECLARE A LOCAL VARIABLE */

      ....

      RETURN 2+Z;     /* RETURN A VALUE HERE */

      ....

      RETURN;         /* RETURN...BUT WITH NO VALUE THIS TIME */

      ....

   END INT_PRC;       /* END OF PROCEDURE INT_PRC */

   ....

END TEST;             /* END OF MAIN PROCEDURE TEST */
```

CHAPTER 4 - EXPRESSIONS AND ASSIGNMENTS

Introduction

An expression is a rule for computing a numerical, logical, or string value. There are four main types of expressions:

Arithmetic
Relational
Boolean
String

There are five operators which are used to form expressions:

Arithmetic	(+, -, *, /, **, MOD)
Relational	(<, >, =, etc.)
Boolean	(AND, OR, NOT, XOR)
Concatenation	(||)
Sub-unit selectors	([s,l])

An assignment is used to change the value of a variable to the value of an expression.

4.1 Arithmetic Operators And Expressions

General Description

An arithmetic expression is a sequence of numerical constants and variables, called operands, and arithmetic operators.

Precise Description

A. Syntax of an Arithmetic Expression

An arithmetic expression is defined recursively. The following rules specify all possible arithmetic expressions:

1) var	2) exp + exp	3) exp * exp
4) con	5) exp - exp	6) exp / exp
7) exp ** exp	8) (exp)	9) function
	10) exp MOD exp	

Where:
"var" is any integer or real variable, delay variable, subscripted variable (chapter 8), or bit selected variable (see section 4.4 below).
"con" is any integer or real constant.
"function" is any real or integer function invocation (chapter 3).

"exp" is any expression - arithmetic, relational, or boolean.

*, +, -, /, and ** are the arithmetic operators and have the usual meanings attached to them:

```
*   denotes multiplication
+   denotes addition
-   denotes subtraction
/   denotes division
**  denotes exponentiation
MOD is the remainder of dividing exp by exp
```

Examples:

4	Constant
ALPHA1	Variable
A23 + 15	Exp + Exp
(B3 + 6) * (ERR - 4)	Exp * Exp
A**2	Exp ** Exp
3.5/(C+6.2)	Exp / Exp
SIN(2*PI)	Function
2 * FUNC1(A)	Exp * Exp
24.3 + MAT(INDEX)	Exp + Exp

B. Operator Precedence

When a legal CRASH expression is evaluated, operations are performed in an order determined by the precedence of the operator. The precedence of arithmetic operators in CRASH, in order of decreasing precedence, is:

```
**          exponentiation
*, /, and MOD  multiplication, division, and remainder
+ and -     addition, subtraction, unary plus and minus
```

Parenthesis can be used to enclose a term which is to be evaluated out of the normal precedence. Parenthesis have the lowest precedence in CRASH. When two operations have the same precedence, the one occurring first (the leftmost one) is evaluated first.

Example:

```
(ALPHA + 2) * 3.0 ** SIN(BETA + 1) + 43.2 / 31
```

The order in which operations are performed in the above statement are indicated by superscripts.

C. Type Conversions

Type conversions are automatically applied by CRASH. When evaluating an operation in an expression the types of the two operands are checked. If both are integer or bit values, the operation is performed using integer arithmetic. If, however,

one or both operands are real, the operation is performed using real arithmetic. The operand which is integer will be converted to real before the operation is performed. Since CRASH evaluates an expression one operator at a time, it is possible to have part of an expression computed using integer arithmetic and part computed in real. If any term in the expression is real the final result will have a real value, although part of the expression may have been computed in integer. Integer expressions can be forced to be evaluated using real arithmetic by adding "0.0" to one of the operands.

4.2 Logical Expressions

General Description

Logical or Boolean expressions are used to make decisions concerning the flow of control in a CRASH program. A logical expression may have one of two possible values - true (least significant bit a one) or false (least significant bit a zero). Relational operators and Boolean operators are used to form a logical expression.

Precise Description

A. Relational operators

A relational expression is a type of logical expression which involves a relational operator. Relational operators perform a value comparison between the two operands. The syntax of a relational expression is:

```
exp rel exp
```

Where:
"exp" is any constant, variable, or expression.
"rel" is one of the following relational operators:

=	is equal to
~=	is not equal to
>	is greater than
~>	is not greater than
>=	is greater than or equal to
<	is less than
~<	is not less than
<=	is less than or equal to

"NOT" may alternately be used instead of the not sign (~). The value of the expression is true if the relation holds true. Character expressions are compared first by length, then character by character if the lengths are equal (the ASCII collating sequence is used). Internally, the relational expression is represented as a BIT(1) variable.

Examples:

```
BIG > SMALL
(2 + 3) > 4
CHAR1 = INBUF
CATS ~= DOGS
```

B. Boolean Operators

Boolean operators are used to perform the logical AND, OR, NOT, and EXCLUSIVE OR functions on CRASH expressions. They are especially useful for combining several relational expressions into a more complex compound conditional. The legal CRASH boolean expressions are as follows:

1)	exp AND exp	2)	exp OR exp	3)	NOT exp
4)	exp & exp	5)	exp | exp	6)	~ exp
7)	exp XOR exp				

Where:
"exp" is any integer or bit constant, variable, or expression. REAL expressions may not be used in a boolean expression and will not be converted to INTEGER.

"AND" and "&" mean to perform the logical AND of the two expressions. "OR" and "|" mean to perform the logical OR of the two expressions. "NOT" and "~" are the logical negation operators. "XOR" is the logical EXCLUSIVE OR of the two expressions. All boolean operators work on a 16 bit value and produce a 16 bit result. Since all BIT variables occupy 16 bits no matter what size they were declared to be this should cause no trouble.

Examples:

```
TYPE AND TYPEMASK
SWITCHWORD | SET_7
(NUM > SMALLEST) & (NUM < LARGEST)
( E(1) | E(2) | E(3) ) & EMASK
CHECK_SUM XOR DATAWORD
(A OR B) & (NOT C)
```

4.3 Concatenation And String Expressions

General Description

An expression can be formed which has a character or string value. The only string operator is concatenation which combines two (or more) strings into a longer string.

Precise Description

The syntax for concatenation is:

```
exp1 || exp2 || ... || expn
```

Where:
"exp" is any CRASH constant, variable, or expression. "exp" can be of any type - integer, real, or character. If the type is numerical, it is converted into an ASCII character string. Real numbers occupy an 11 character field, integers take up a 6 character field. If the whole field is not needed, blanks are padded on the left. If a concatenation results in a string longer than 255 characters, an ERROR trap is made to the operating system, and a message is printed to that effect. The program can be restarted from that point and the concatenated string will appear to have been truncated at 255 characters.

Examples:

In the following example, several constant strings are concatenated with some date information in integer form.

```
'TODAY IS ' || DAY || ', ' || MONTH || ' '
   || DATE || ', ' || YEAR || '.';
```

If DAY='FRIDAY', DATE=26, MONTH='AUGUST', AND YEAR=1977, the result of the concatenation is a string with the following value:

```
TODAY IS FRIDAY, AUGUST     26,   1977.
```

4.4 Sub-unit Selection

General Description

CRASH also has a way to reference, or to assign values to, individual bits of an integer or bit variable, or individual characters of a character variable.

Precise Description

Individual bits or characters in a variable may be referenced. A field description is used to specify which bit or character starts the field, and how many are in the field. A variable which has a field description appended to its name is considered to be a single variable, and may be used in expressions, etc. like a simple variable. The syntax is:

```
var [start]     or     var[start,length]
```

Where:
"var" is any simple bit, integer, or character variable.
"start" is the number of:

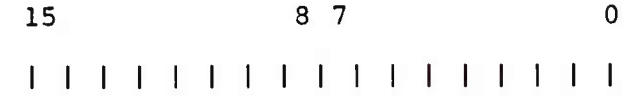
1) The first character in the sub-unit
2) The first bit in the subunit

"length" is the number of:
1) Characters in the field
2) Bits in the field.

"start" and "length" can be any integer or bit constant, variable, or expression.

Examples:

```
LOW_BYTE = BITS[0,8];
HIGH_BYTE = BITS[8,8];
HIGH_BYTE = BITS[8];
OUTBUF[16,6] = DATE;
COMMAND = INBUF[0,BREAKCHAR-1];
```



4.5 Assignment Of Values

General Description

An assignment statement is used to change the value of a variable to some new value specified by a CRASH expression.

Precise Description

A. The Assignment Statement

An assignment involves the use of the assignment or substitution operator. The syntax of an assignment statement is:

```
var = exp;

   or

var1,var2,...,varn = exp;
```

Where:
"var" is the variable being assigned to.
"exp" is any type compatible CRASH expression.

The value of the expression is computed. Any necessary conversions are made, and the new value of "var" is the value of the expression.

B. Type Conversions

If the value of "exp" is a character string, the variable must be of character type or an error will occur. In all other cases, the value of the expression will automatically be converted to match the type of the variable. Assignment of a REAL value to an INTEGER variable will result in a truncating conversion. A rounding conversion is done by adding "0.5" to the value before the conversion.

C. Multiple Assignments

If a list of variables is specified (the second form in A above), all of the variables are assigned the value of the expression. The list of variables is processed from right to left. The variables must be of compatible types. Consider the following:

```
 <--- Processing Order
A,B,C = 6;
```

A, B, C can be either real or integer, and no problems will occur. If C is character, the expression will be converted to character and assigned to C. A and B must then also be character, or an illegal conversion will be recognized. By suitable ordering, these problems can be avoided. If the order had been "C, A, B" then the final result will be converted to character after the assignment to A and B, and no error will occur.

Examples:

```
B = 7;
CHAR1 = 'THIS IS A TEST';
CHAR2, NUM1, NUM2 = 0.0;
TABL(2) = VAL;
RESULT = A + B + C + (4**EXPO);
SINEBETA = SIN(BETA);
```

4.6 Complete Operator Precedence

General Description

A complete list of the precedence of operators in CRASH is given below. Parenthesis have the lowest precedence and are used to override the normal precedence of operations.

	functions, subscripts, delay specifications, sub-unit selections
	**
Arithmetic	*, /, MOD
	+, -
Concatenation	||
Relational	<, >, =, <=, >=, ~=, ~>, ~<
	NOT, ~
Boolean	AND, &
	XOR
	OR, |
Assignment	=
	( )

CHAPTER 5 - CONTROL CONSTRUCTS

Introduction

The actual work done by a program is performed by assignment and I/O statements. Other statements, called control constructs specify the structure and order of execution of the statements.

The following control constructs are included in the definition of CRASH:

```
DO;
DO cvar = init TO final BY inc;
DO cvar = exp1, exp2, ..., expn;
DO WHILE exp;
DO UNTIL exp;
DO CASE exp;
EXIT DO;
EXIT DO label;
NEXT DO;
NEXT DO label;
IF exp THEN statement
IF exp THEN statement ELSE statement
GO TO label;
```

5.1 DO Construct

General Description

The DO construct performs three functions in CRASH. The first is repetitive execution of a sequence of statements. The second is selection of one of several statements for execution. Finally, a "DO" is provided which allows a group of statements to be considered as one statement.

Precise Description

A. Levels, Nesting and Scope

In CRASH, each DO group is considered as one single statement, although there may be many statements inside the group. The statements that comprise the group are nested one level higher than the statements in which the DO group appears. A DO statement can appear inside a DO group, so DO statements can be nested to any level.

B. Types of DO statements

There are 6 types of DO statements in CRASH. In the following descriptions, "body of group" is any sequence of CRASH statements.

B.1 The simple DO

The simple DO is used whenever a sequence of statements needs to be considered as a single statement. The uses will become clearer in the section on IF...THEN...ELSE below. The form is:

```
DO;
   statement1
   statement2   |
   ...          |   body of group
   statementn
END;
```

Where:
"statement" is any CRASH statement.
Note that the whole DO group is itself considered to be a single statement on the level in which it appears.

B.2 The iterated DO

The iterated DO is used to execute a sequence of statements repeatedly for values of a control variable. There are two forms of iterated DO:

```
DO cvar = init TO final;
   body of group
END;
```

and

```
DO cvar = init TO final BY inc;
   body of group
END;
```

Where:
"cvar" is the variable to be used to control the iterations.
"init" is the initial value assigned to "cvar" before the first execution of the loop.
"final" is the value against which "cvar" is tested to terminate execution of the loop.
"inc" is the amount to add to "cvar" after each execution of the loop.
"init", "final", and "inc" can be any constant, variable or expression which can be evaluated to yield an integer. "cvar" may be either an integer or real variable.

If "BY inc" is not specified, "cvar" is incremented by one each time.

Execution of the loop occurs in the following order:

1) "init" is evaluated and assigned to "cvar"
2) "cvar" is compared with "final"
3a) Variable with positive value or positive constant

"inc":
If "cvar" is greater than "final", execution of the loop terminates.
3b) Variable with negative value or negative constant "inc":
If "cvar" is less than "final", execution of the loop terminates.
4) The body of the group is executed.
5) "inc" is added to "cvar".
6) continue with 2).

B.3 The Stepped DO

The stepped DO is like the iterated DO, except that the loop is executed only for specific values of the variable. The form is:

```
DO cvar = exp1, exp2, ..., expn;
   body of group
END;
```

Where:
"cvar" is the variable which will take on the values specified in the list of expressions.
"exp" is any CRASH constant, variable, or expression. The expressions will be converted to integer before assignment to "cvar".
The loop is executed once for each value in the list. "cvar" will take on the first value the first time through the loop, the second value on the second time, and so on until the list is exhausted. Execution then continues with the first statement after the END.

B.4 The DO WHILE

The DO WHILE is used when a sequence of statements is to be executed as long as some condition remains "true". The form is:

```
DO WHILE expression;
   body of group
END;
```

Where:
expression is any expression in CRASH which can be evaluated to yield a true (least significant bit a one) or false (least significant bit a zero) boolean value. If the expression has a real value (for some peculiar reason), it will be converted to integer before its least significant bit is tested.
The DO WHILE group is executed in the following order:

1) The expression is evaluated.
2) If it has a false value(least significant bit = 0), execution of the group terminates.
3) The body of the group is executed.

4) Continue with 1).

It is therefore possible that the loop will never be executed, if the expression has an initially false value.

B.5 THE DO UNTIL

The DO UNTIL is used when a sequence of statements is to be executed until some condition is true. The form is:

```
DO UNTIL expression;
   body of group
END;
```

The DO UNTIL group is executed in the following manner:

1) The body of the group is executed
2) The expression is evaluated.
3) If the expression is true(LSB.= 1), execution of the group is terminated.
4) Continue with 1).

The DO UNTIL loop is always executed at least one time.

B.6 The DO CASE

The DO CASE is one of the most powerful programming tools available in CRASH. With it, the user has the ability to perform an n-way branch. The form is:

```
DO CASE expression;
   statement0
   statement1
   ...
   statementn
END;
```

The value of the expression is evaluated. If the value is zero, statement0 is executed. If the value is n, statementn is executed. The expression must be integer in type. If the value of the expression is less than zero or greater than "n", no statement in the group is executed.

C. Identifiers and DOs

The DO statement can be labelled like any other statement. The form is:

```
label: DO ...
```

Where:
"label" is any unused identifier.
If a label is specified on the DO statement, it must also appear

on the END for that group in the following way:

```
END label;
```

The advantages of labelling DOs are that each time an END with a label is encountered, a check is made to see that no ENDs have been left out, or that too many have been used. This is very helpful for the inexperienced user who will be apt to mismatch DO-ENDs, especially if the code is not indented to show the beginning and end of the groups.

D. EXIT and NEXT DO

It is often desirable to be able to exit from a DO group before its normal completion. When this is the case, the following statement can be used:

```
EXIT DO;
EXIT DO label;
```

Execution of this statement will cause control to be transferred to the first statement after the END of the current DO group if no "label" was specified. If a "label" was specified, control is transferred to the first statement after the END of the labelled DO group.

Sometimes it is convenient to terminate a particular iteration of a DO group, and continue with the next iteration. In this case, the statement:

```
NEXT DO;
NEXT DO label;
```

can be used. When this statement is executed, control continues as if the END statement for the DO group had been there. If the condition for executing the DO group is still satisfied, the next iteration is performed in the usual manner. If a "label" was specified on the NEXT DO statement, the next iteration of the labelled DO group is started.

Note that the EXIT DO and NEXT DO statements can be used to eliminate GO TO statements from a structured program. They are not, however, a substitute for good structured code using only the above DO constructs and the IF...THEN statement (discussed later).

Examples:

An example of iterated DO is to sum the integers from 1 to 10.

```
SUM = 0;
SUM_IT:  DO I = 1 to 10;
   SUM = SUM + I;
END SUM_IT;
```

This time, only even integers are to be summed:

```
SUM = 0;
SUM_EVEN:   DO I = 2 TO 10 BY 2;
   SUM = SUM + I;
END SUM_EVEN;
```

A chemical is being titrated with an indicator which changes color depending on the pH of the solution. A sensor is set up which determines the color of the solution and sends this data as an integer specifying one of five colors to the computer. It is the computers job to control the valves which allow more acid or base to be mixed with the chemical. The following DO CASE might be used to decide what action is to be performed based on the color of the solution:

```
TITRATE:   DO CASE SOLUTION_COLOR;

/* CASE 0 */
   /* ADD ACID AT FAST RATE */
   CALL ADD_ACID(1);

/* CASE 1 */
   /* ADD ACID AT SLOW RATE */
   CALL ADD_ACID(0);

/* CASE 2 */
   /* ALL DONE */
   DO;
      /* TURN OFF FLOW IF NECESSARY */
      IF ACID_IS_ON THEN
         CALL STOP_ACID;
      ELSE IF BASE_IS_ON THEN
         CALL STOP_BASE;
   END;

/* CASE 3 */
   /* ADD BASE AT SLOW RATE */
   CALL ADD_BASE(0);

/* CASE 4 */
   /* ADD BASE AT FAST RATE */
   CALL ADD_BASE(1);

END TITRATE; /* DO CASE */
```

Some data are being computed using the prime numbers from one to thirty. A stepped DO can be used to do this efficiently:

```
DO PRIME = 1,3,5,7,11,13,17,19,23,29;
   VAL = VAL + (2 * PRIME);
END;
```

5.2 IF...THEN and IF...THEN...ELSE

General Description

There are two forms of the IF statement in CRASH. The IF...THEN statement allows conditional execution of a CRASH statement. The IF...THEN...ELSE is the same except that an alternate statement is executed if the condition is not met.

Precise Description

A. IF...THEN

The syntax of the basic IF statement is:

```
IF exp THEN statement
```

Where:
"exp" is any CRASH expression which can be evaluated and converted (if necessary) to a Boolean (True/False) value (see section 7.3).
"statement" is any single CRASH statement. A group of statements can be used if they are enclosed by DO;...END; (see section 5.1).

The expression is evaluated. If the value is true (least significant bit=1) the "statement" is executed.

Examples:

```
IF PWRON & ((DANGER_CONDS & DMASK) = 0) THEN
   CALL START_UP;

IF STOCK > DEMAND THEN
   PRICE = LOW_PRICE(ITEM);
```

B. IF...THEN...ELSE

The ELSE clause allows alternate action to be taken if the condition is not met. The syntax is:

```
IF exp THEN basic statement
ELSE statement
```

Where:
"exp" is any CRASH expression which can have a boolean value.
"basic statement" is any CRASH statement except a scheduling statement or another IF statement.
"statement" is any CRASH statement.

The expression is evaluated and converted to a Boolean type if necessary. The least significant bit is tested and if true the basic statement is executed. If the value is false, the statement is executed. One and only one of the statements is

executed.

The requirement that a basic statement follow the THEN is to avoid ambiguity in CRASH. Any statement can be made basic by imbedding it in DO;...END;.

Examples:

```
IF (STATUS & SMASK) = 3 THEN
   ESTAT = TRUE;
ELSE ESTAT = FALSE;

IF COMMAND = 'YES' THEN
   YES:   DO;
      ...                          /* LEAVE OUT DETAILS */
   END YES;
ELSE
   NO:   DO;
      ...                          /* MORE DETAILS */
   END NO;
```

C. Nesting IF Statements

If the statement following the THEN or ELSE is another IF statement, the conditional statement is said to be nested. Care must be exercised when using nested IF statements as syntactically correct statements may not perform the desired control flow. One way to minimize the possibility of this is to use a consistent form of indenting throughout the CRASH program. If indenting is used uniformly, the programmer will have a much easier time constructing the flow of control and will make the flow of control more readily apparent during the testing of the program. Consider the following IF statement. It is syntactically correct but probably won't work the way it was intended because of incorrect indenting.

```
IF exp THEN
   IF exp THEN
      s
ELSE
   IF exp THEN
      s
   ELSE
      s
```

Simply indenting control statements will not change the flow of control (that is determined by the syntax only) but it will help in understanding the flow of control. The rule to use is that an ELSE belongs with the last preceding IF statement which does not have an ELSE. So in the above example, the ELSE which looks like it belongs with exp really belongs with the IF on exp. A correct way to indent these statements is:

```
IF exp THEN
   IF exp THEN
      s
   ELSE
      IF exp THEN
         s
      ELSE
         s
```

Examples:

```
IF NUM > 7 THEN
   IF ITEM <= BIGITEM THEN
      BOX = BIG;
   ELSE
      BOX = CRATE;
```

5.3 GOTO And GO TO

General Description

The GO TO statement is used to transfer control, either conditionally or unconditionally, to another segment of the program.

Precise Description

A. Transfer of Control

Any CRASH statement can be labelled with an identifier in the following way:

```
label: statement
```

Where:
"label" is any unused identifier.

The GO TO statement has the following forms:

```
GOTO label;
GO TO label;
```

Execution of a GO TO will cause a jump or branch to the statement labelled with "label". This statement will be the next statement to be executed.

A GO TO can be made conditional by using it with an IF statement:

```
IF expression THEN GO TO label;
```

If the expression has a true value, control is transferred to the statement with "label".

B. Restrictions on use

GO TO statements should not be used to jump into the middle of an iterated DO group or to jump into another procedure. Unless the internal workings of CRASH are known to the programmer the results can be unpredictable, and will likely result in a non-functioning program.

The GO TO is not considered to be a structured programming tool. In almost every situation, the GO TO can be eliminated by thoughtful use of the other control constructs described so far.

There are many reasons why the GO TO should not be used. Structured programs read from top to bottom. If GO TOs are used, very often the reader will find himself flipping back and forth through a long listing trying to find out "Where do I go from here?". Another reason is that GO TOs get in the way of any formal attempt at proving the "correctness" of a program.

Several studies have been undertaken to determine the productivity of programmers using a structured programming language versus an unstructured language such as FORTRAN. Rough figures indicate that two to three times as many debugged statements per day can be produced using a structured language. According to Yourdon:

> Though no experimental evidence has yet been gathered, there is good reason to believe that the increase in productivity will be even higher in the area of real time systems (operating systems, process control systems, etc.) than with the non-real-time application programs.

GO TOs should only be used when the structured approach is even more awkward and confusing, or if it will clearly use a good deal less memory or execution time and still be understandable to someone else.

Examples:

A reasonable use of the GO TO is to avoid duplication of a small group of statements.

Yourdon, Edward, Techniques of Program Structure and Design, Prentice-Hall

```
   CONTROL:   DO CASE KNUM;

   /* CASE 0 */
      GO TO SHUT_DOWN;

   /* CASE 1 */
SHUT_DOWN:
      DO;
         CALL SAVE_STATUS;
         CALL POWEROFF;
         OUTPUT = 'POWER HAS BEEN SHUT OFF';
      END SHUT_DOWN;

   /* CASE 2 */
      CALL INIT;

   ...

   END CONTROL; /* DO CASE */
```

CHAPTER 6 - TASKING AND TIMING

Introduction

It is sometimes desirable to have many different activities take place concurrently within the computer. Normal procedure calls cause suspension of the calling program until the called program has RETURNed. CRASH allows the possibility of having several procedures active at one time, without requiring the completion of one before another can execute. Such procedures, which can 'live' independently of other procedures, are called tasks.

There are a variety of ways a task can be scheduled to execute. It can be synchronized with the clock, with the procedure which first invoked it, or with some other procedure. It may even be scheduled to execute asynchronously (triggered by some external event or I/O completion).

Besides having a scheduling attribute, a task also has an attached attribute called its priority, which specifies its importance and timing requirements in the collection of programs being executed.

The six basic scheduling statements which are a part of the CRASH language are:

```
AT <time> START <task>;
IN <time> START <task>;
EVERY <time> START <task>;
ON <condition> START <task>;
START <task>;
CANCEL <task>;
```

The meaning and use of the above statements is discussed in the following sections.

6.1 Timing

General Description

Since a task may be scheduled to start with certain timing constraints, there must be a way to specify these constraints in a CRASH program.

Precise Description

A time may be specified in several different ways: minutes, seconds, milliseconds (thousandths of a second) or in 100-microsecond units (millionths of a second). The number of units of time specified may be stated as a constant (INTEGER or REAL) or as a variable. If the number of units is stated as a variable, this variable must not be an array element, an element

of a delay variable (ANALOG or DISCRETE), nor the value of a function invocation.

Examples:

The following are all legal ways of specifying a time.

3 MIN	(3 minutes)
5.25 SEC	(5.25 seconds)
XYZ MSEC	(variable number of milliseconds)
IJK	(variable number of 100-usec units)

The following are all illegal attempts to specify a time.

'3' MIN	(units cannot be character valued)
3 MINUTES	(MIN is the proper way to specify the magnitude)
XYZ SECONDS	(SEC is the proper way to specify the magnitude)
IJK(3) MSEC	(Array elements are not permissible unit specifications)
SIN(PI)	(Function invocations are not permissible unit specifications)

In the discussion that follows, the symbol <time> will be used to denote a legal timing specification.

6.2 Declaring A Task

General Description

A task is declared in CRASH just like an external procedure except that it is preceded by the word TASK instead of the word ROUTINE in the type specification for the procedure.

Precise Description

Every task must be declared in a scope which surrounds the statement(s) in which it is refered to. In the declaration section the statement:

```
TASK TNAME;
```

must appear if TNAME is to be referred to within that scope. This means that TNAME must be declared in the current level procedure, or a procedure on a lower level which contains the current procedure.

6.3 Defining A Task

General Description

A Task is defined in CRASH just like an external procedure except that it may not have any parameters and must be declared slightly differently in the procedure heading.

Precise Description

A task is defined by writing:

```
TNAME: PROCEDURE TASK;

   ... procedure body;

END TNAME;
```

6.4 Task Identifiers

General Description

In order to refer to a currently executing task, it is necessary to keep a unique name for every invocation of a task. Task Identifiers are variables to which are assigned such unique names.

Precise Description

The same procedure may be scheduled to execute several times from within a system of programs. Just giving the name of the procedure which is to execute, therefore, does not uniquely specify which invocation of the task is meant when it is referred to. To alleviate this problem, whenever a task is scheduled, a variable must be surrounded by parenthesis and appended to the task name. The scheduler will return a unique identifier for that particular invocation. In CRASH, this is done by saying TASKNAME(variable) . The reasons why a previously invoked task might have to be referred to will become clearer in later sections.

The task identifier for the MAIN procedure is a pre-defined integer variable MAINID.

6.5 Priority

General Description

Every invocation of a task must specify a priority with which the invocation should be completed. The higher the priority in relation to other tasks currently executing, the sooner the newly invoked task will be completed.

Precise Description

A priority specification is an integer value between 1 and 250. It may be specified as either a constant or a variable. If it is specified as a variable, it must not be an array element, element of a delay variable, sub-unit of a variable, MAP variable, or a function invocation. The task that is invoked is entered into the collection of tasks competing for execution. Since only one task invocation may proceed at any time, the one which is selected to proceed is the one with the highest priority. If a task with a lower priority than the one which has just been invoked has been proceeding, it is pre-empted until the higher priority task has completed execution. There is no time slicing or sharing done by the operating system scheduler during I/O waiting. The highest priority task which is in competition for CPU time will run to completion before a lower priority task can proceed.

The MAIN procedure and any subroutines or functions it calls run with a priority of 10 (see the OSWIT Users Manual).

Examples:

The following are examples of specifying a task and its priority.

Example:

```
TNAME(X) PRIO(2)
TNAME(QSL) PRIO(I)
```

Given tasks T1, T2, and T3, with priorities 10, 20, and 30 respectively, and supposing that T3 is currently proceeding in execution, the invocation of task T4 will cause the sequence of task completions to be as follows (assuming no other tasks are invoked).

Example:

```
T4 PRIO(50) : T4, T3, T2, T1

T4 PRIO(30) : T3, T4, T2, T1

T4 PRIO(25) : T3, T4, T2, T1
```

6.6 Scheduling A Task

General Description

A task is scheduled to execute by a special statement called a Scheduling statement. Every time a task is mentioned in one of these scheduling statements, a new invocation of the task is scheduled (Thus it is possible to have a procedure executing concurrently with itself).

If this is done, however, care must be taken to make the task re-entrant and use all automatic variables in the task.

Precise Description

A. AT

A task may be scheduled to start at a certain time:

```
AT <time> START TNAME(I) PRIO(10);
```

In this case <time> refers to the number of milliseconds, seconds, or minutes after midnight.

B. IN

It may be scheduled to start at a certain time increment from the instant at which it was scheduled:

```
IN <time> START TNAME(ID) PRIO(J);
```

C. EVERY

It may be scheduled to start periodically:

```
EVERY <time> START TNAME(Z) PRIO(I);
```

D. ON

It may even be scheduled to start execution upon the occurrence of some event (called a condition) which occurred externally to the procedure which is scheduling it:

```
ON <condition> START TNAME(ID) PRIO(75);
```

Such external conditions will be discussed in sections 6.8 and 8.5.

E. START

A task may simply be put into the queue of tasks competing for execution immediately:

```
START TNAME(ID) PRIO(116);
```

Examples:

To start a task at 3 minutes (on the computer's clock) past midnight
Example:

```
AT 3 MIN START BLOW_WHISTLE(ID1) PRIO(10);
```

To start a task in X seconds from now
Example:

```
IN X SEC START APPLY_BRAKES(ID2) PRIO(19);
```

To start a task every 15 milliseconds

Example:

```
EVERY 15 MSEC START CHECK_LEVEL(ID3) PRIO(12);
```

To start a task upon the occurrence of an interrupt from a logical device

Example:

```
ON INTERRUPT_A(LDN) START CHECK_SWITCHES(K) PRIO(95);
```

To start a task immediately:

Example:

```
START SAMPLE_STARTER(ID4) PRIO(129);
```

6.7 Cancelling A Task

General Description

A task may be cancelled if its purpose has been fulfilled and is no longer required.

Precise Description

The following scheduling statement

```
CANCEL TNAME(variable);
```

will cause the task to be cancelled. If the task is currently proceeding or has been pre-empted by another higher priority task during its execution, it will be allowed to complete its execution. If, however, it has not yet begun execution, it will be cancelled before and never be allowed to proceed. If it is desired to re-schedule the task, that is permitted, but it will occur as if it had never been scheduled before.

All tasks should be cancelled when they are no longer needed. Each task that has been scheduled requires that a special block of storage be maintained by the scheduler which defines its priority, etc. There are a limited number of these task control blocks available (specifically 255 with the OSWIT scheduler).

Examples:

The following is a legal task cancellation

Example:

```
CANCEL TNAME(I);
```

The following is an illegal task cancellation since no task identifier was specified.

Example:

```
CANCEL TNAME;
```

Operating System WIth Trains. Developed at the University of Michigan.

6.8 Conditions

General Description

This is the second section of the manual dealing with conditions, the first appearing in the section on CONTROL CONSTRUCTS. It should be emphasized that the two types of conditions (those which cause the execution of a single CRASH statement, and those which can cause the invocation of a task) are essentially different (see section 8.5). The only conditions discucsed here are those relating to input-output devices and their (possibly) asynchronous behavior.

Precise Description

Two conditions which may cause a task to be invoked are IO_RETURN and INTERRUPT. IO_RETURN occurs when an input-output unit signals the computer that an I/O operation has completed on a specified unit with a particular return code. Possible return codes can be end-of-file, end-of-disk, or successful I/O completion. INTERRUPT can occur for a variety of reasons, and those reasons vary from device to device. The possible INTERRUPT reasons are dependant upon the device attached to the logical unit. OSWIT users should consult the command description section in the OSWIT manual.

A. IO_RETURN

When it is desired to start a task because of a particular return code on one (or more than one) device, the statement:

```
ON IO_RETURN(rc,ldn1, ldn2, ... , ldni) START TNAME(ID) PRIO(M);
```

should be used. The parenthesized list (ldn1, ldn2, ... , ldni) is intended to represent a list of logical device names corresponding to the actual input-output units. "rc" is the return code from an I/O operation which is to cause the task to be invoked. The return codes which can occur on the various devices are listed in the OSWIT users manual. Several macros (see chapter 10) are available for the common rc's (i.e. ENDFILE, IO_SUCCESS, ENDDISK).

The IO_RETURN condition is an alternative to waiting for an I/O operation to be completed. It can be used to allow simultaneous I/O activity and processing.

B. INTERRUPT

Associated with each device are two possible interrupts, A and B. When it is desired to start a task because of an INTERRUPT condition on a device, the statements

```
ON INTERRUPT_A(ldn) START TNAME(ID) PRIO(M);
```

```
ON INTERRUPT_B(ldn) START TNAME(ID) PRIO(M);
```

should be used. The OSWIT users manual specifies the possible conditions and associated interrupts. The INTERRUPT condition is typically used with devices that may be activated asynchronously (i.e. a level sensor which triggers when the level of liquid in a tank falls below a certain level).

Examples:

```
ON IO_RETURN(4,SCARDS) START WRITEREPORT PRIO(1);
ON INTERRUPT_A(15) START TRIPSWITCH(X) PRIO(4);
ON INTERRUPT_B(10) START UPDATE_SPEED(ID1) PRIO(25);
```

CHAPTER 7 - INPUT AND OUTPUT

Introduction

There are three forms of I/O supported by CRASH which simplifies the problems encountered with more complicated format structures. The first form discussed is used primarily for communicating with the console device and human operators. A second form is used to send and receive data from external devices such as A/D and D/A converters. The third form is used when doing record I/O with floppy disk files, MTS, the console, or any device which supports record I/O.

Since CRASH runs in a real-time environment, most of the I/O statements only start the I/O operation. Two methods are provided to determine when an I/O operation is complete. The wait statement simply suspends the currently executing task until the operation is done. Alternately, an ON-condition can be set up to start a task on a specified return code from the operation. This enables the currently executing task to continue executing during the I/O.

7.1 INPUT

General Description

The identifier INPUT is used to read character, integer and real constants from SCARDS. In conjunction with the console input buffer, each reference to INPUT will convert one constant from SCARDS. SCARDS is defined as being logical unit 26 in the OSWIT operating system.

Precise Description

INPUT is a pseudo-variable that may appear anywhere a variable can, except that it may not be on the left side of an assignment. Another restriction on the use of INPUT is that sub-unit selectors cannot be used. Whenever INPUT is referenced a constant (as defined in section 2.1) is read from SCARDS. Several constants may be on each line, separated by commas and/or one or more blanks. Initially the console input buffer is empty. The first occurrence of INPUT will cause the user to be prompted for a line of input. This line can have one or more constants on it. The first constant will be used for this input. The next time INPUT is used, the input buffer will be checked to see if it is empty. If it is, another line will be read from SCARDS. If the buffer is not empty, the next constant in the buffer which hasn't yet been used will get used.

The type of constant that INPUT is expecting is determined

by the context around INPUT. If it appears in a concatenation or a simple replacement of a character variable the value expected will be a string constant. If INPUT appears in a simple integer replacement, or in an integer arithmetic expression, an integer constant will be expected. Otherwise, a real or integer value can be used. Note that string constants are enclosed by primes (section 2.1D). If the type of constant does not match the type expected, or neither a blank or comma separates two constants the user will be asked to reinput the line from the point of error. INPUT is intended to be used conversationally from the console device and automatically waits for the operation to be completed.

Examples of legal uses of INPUT:

```
REAL_VARIABLE = INPUT;
INTEGER_VARIABLE = INPUT;
IF INPUT = 'STOP' THEN RETURN;
```

Example of illegal use of INPUT:

```
INPUT = ALPHA;
```

7.2 CARD

General Description

The identifier CARD is used to read a complete line from SCARDS.

Precise Description

CARD is a character pseudo-variable which can appear anywhere a variable can except the destination of an assignment. Whenever CARD is referenced, a line is read from SCARDS and used as the value of CARD. As with INPUT, CARD automatically waits for the input operation to be completed. Note that CARD has no effect on the console input buffer. Care must be taken with CARD when it appears in a statement like the following:

```
CHARVAR = CARD;
```

If the length of the next line from SCARDS is longer than the maximum specified for CHARVAR, part of the program may be destroyed. Since the maximum length of an input line is 255, a size of 255 will avoid trouble here.

Example:

```
COMMAND_LINE_BUFFER = CARD;
```

7.3 OUTPUT

General Description

The identifier OUTPUT is used to write a line to SPRINT. SPRINT is defined as logical unit 28 under the OSWIT operating system.

Precise Description

OUTPUT behaves exactly like a simple character variable, except that it must always be the destination of an assignment. Substring assignment cannot be performed on OUTPUT. Conversions from numerical types to character form are automatically applied. Integer variables require a six character field, reals take up an eleven character field. No leading or trailing blanks are included if the number takes up the whole field. If the whole field is not needed, blanks are padded on the left. Each time an assignment is made to OUTPUT, a line is written on SPRINT.

OUTPUT only queues the line for printing on SPRINT. This allows processing to continue during an output operation. At some point before the program terminates a WAIT statement should be executed to make sure all output has stopped. Another important consideration when using OUTPUT is the limited buffer space available from the operating system. If too much output has been queued (OUTPUTing without WAITing) the operating system may run out of buffers. This error is usually fatal, but generally occurs only when SPRINT is assigned to the console device (the default) since it is a slow device and output can queue up rapidly for it.

Examples of legal uses of OUTPUT:

```
OUTPUT = INTEGER_VARIABLE;
OUTPUT = 'ENGINE ' || ENGINE_NUM || ' HAS DERAILED';
OUTPUT = REAL_VARIABLE;
```

Examples of illegal uses of OUTPUT:

```
ALPHA = OUTPUT;
INPUT = OUTPUT;
OUTPUT[0,5] = BETA;
```

7.4 GET

General Description

The GET statement is used to input data from an external device.

Precise Description

The GET statement has the following form:

```
GET varlist;
```

Where:
"varlist" refers to either a single variable, or a list of variables separated by commas. The variables must be either DISCRETE or ANALOG variables, and must have at least an LDN specified in their declarations. Depending on whether WORD or BYTE was specified in the declarations, an 8 or 16 bit value is read from the specified LDN.

The GET statement automatically waits for the input to complete.

A. Analog Real

The value read is converted to real, divided by the scale factor, and the offset subtracted. The equation for this operation is:

```
VALUE = (INVAL / SCALE) - OFFSET.
```

If a DELAY has been specified for this variable, the value becomes the new current element.

Example:

```
GET SERVO_POSITION;
GET CAR_VEL,THROTTLE;
```

B. Analog Integer

Identical to analog real except the value is converted back to a 16-bit integer before it is stored.

Example:

```
GET VOICE_SAMPLE;
```

C. Discrete

The value is stored as the new current element. No scale or offset are applied.

```
GET PHOTOCELL_STATUS;
```

7.5 PUT

General Description

The PUT statement is used to write data to external devices other than the console.

Precise Description

The PUT statement has the following form:

```
PUT varlist;
```

Where: "varlist" refers to a single variable, or a list of variables separated by commas. The variables must be either DISCRETE or ANALOG variables, and must have at least an LDN specified in their declarations. The final quantity written on the LDN is either an 8 or 16 bit integer, depending upon whether the BYTE or WORD attribute was specified.

The PUT statement only initiates an output operation. A WAIT must be executed (see section 7.8) if the output must finish before further execution of the current task can occur.

A. Analog Real

The offset is added to the variable. This value is then multiplied by the scale factor. The result is converted to an 8 or 16 bit integer before being written on the LDN. The equation for this operation is:

```
OUTVAL = SCALE * (VALUE + OFFSET)
```

If the output value is less than the low clamp value or greater than the high clamp value after scaling and offseting, the value is clamped at the minimum or maximum value respectively.

Example:

```
PUT IDLE_MIXTURE;
```

B. Analog Integer

The value of the variable is converted to real. The offset value is then added, and it is multiplied by the scale factor. The resulting value is converted back to an 8 or 16 bit integer and written on the LDN. The value is clamped if necessary as in A above.

Example:

```
PUT FUEL_VEL,AIR_VEL;
```

C. Discrete

The value is directly output to the LDN as an 8 or 16 bit integer.

Example:

```
PUT SWITCH_SETTINGS;
```

7.6 GET RECORD

General Description

A record is a sequence of bytes which are handled as a unit. It is usually much more efficient for a processor to handle many bytes at once during an I/O operation to record oriented devices such as the console or floppy disk.

The GET RECORD statement is used to do record I/O with data or character strings. It can be used with arrays or delay variables.

Precise Description

GET RECORD is used to transfer many bytes of data to a CRASH delay variable, array, or character variable in an efficient manner. The syntax is:

```
GET RECORD(ldn) varlist;
```

Where:

"ldn" is the ldn for the I/O operation.

"varlist" is a list of delay variables, scalar character variables, and/or arrays. Up to 255 bytes (127 words) may be transferred to the variable's storage area starting with the first element location. No conversions are performed; the data are simply copied byte by byte into the array, character variable, or delay variable from the specified ldn. Only the first 255 bytes of a variable can be filled with the GET RECORD statement. The actual amount of data transferred is dependant upon which record is read.

The GET RECORD statement initiates an input operation on the specified logical unit. Either a WAIT statement (see section 7.8) or an IO_RETURN ON condition can be used to determine when the operation has finished.

When GET RECORD is used with a delay variable, the circular list structure is filled beginning with the first storage location. Byte by byte the data are transferred to the circular list structure in order until the record is exhausted. The current element pointer is set to point to the last complete word transferred to the delay variable. In this way previously stored data (on a floppy disk, for example) can be loaded into a delay variable with a single statement.

Examples:

```
GET RECORD(10) DATA_ARRAY;
GET RECORD(11) FLOPPY_BUF;
```

7.7 PUT RECORD

General Description

The PUT RECORD statement is used to do record output with data or character strings. It can be used with arrays, delay variables, or character variables.

Precise Description

PUT RECORD is used to transfer large amounts of data from a CRASH delay variable, character variable, or array to an output device such as a floppy disk for buffering. The transfer is made byte by byte without conversion and is a fast way to move data under the constraints of a real-time system. The syntax is:

```
PUT RECORD(ldn) varlist;
```

Where:
"ldn" is the logical device number to use for output
"varlist" is a list of delay variables, scalar character variables, and/or arrays to output to the ldn. For each variable in the list, a record is written to the specified LDN.

The length of the record written is dependent upon the variable to be PUT. For arrays, the length is the number of bytes in the array, or 255, whichever is smaller. For character variables the length is the current length of the variable. For delay variables, the length is the number of bytes from the first storage location for the circular list to the current element pointer. In addition, for delay variables only, after the PUT RECORD the current element pointer is reset to the first storage location in the circular list. The PUT RECORD, GET RECORD and DELAYFULL constructs can be used to "fill" and "empty" a delay variable in a data acquisition application (see section 8.5).

The PUT RECORD statement queues output for the logical unit. Either a WAIT statement (see section 7.8) or an IO_RETURN condition (see section 6.8) can be used to determine when the output is completed.

Examples:

```
PUT RECORD(5) TEMPERATURE_DATA;
PUT RECORD(6) HORIZ_MOTION,VERT_MOTION;
```

7.8 Waiting For I/O Completion

General Description

I/O operations typically take a long time as far as the CPU is concerned. Therefore, some CRASH I/O statements only initiate I/O operations. It is sometimes necessary, especially on input, to wait for an I/O operation to be completed before continuing execution of the current task. The WAIT statement does this.

Precise Description

The WAIT statement causes suspension of the current task until an I/O return occurs from a device. The syntax is:

```
WAIT FOR ldn,ldn,ldn,... ;
```

Where:
"ldn" is a logical device number to wait for.

When a return occurs from any of the specified ldn's, execution of the task continues. The predefined variable "RETURNCODE" will contain the return code from the device which finished. "UNITNUMBER" is a predefined variable containing the logical unit number of the device which finished.

INPUT, CARD, and GET do an impliicit WAIT, no WAIT statement is needed after them to insure the I/O has finished. OUTPUT, PUT, PUT RECORD, and GET RECORD do not wait. If these operations must be complete before the task can proceed, a WAIT must be issued. Another possibility is to use an ON-condition (see section 5.8).

Examples:

```
WAIT FOR SCARDS; /* SCARDS = 26 */
WAIT FOR 10,11,12;
```

CHAPTER 8 - ARRAYS AND DELAY VARIABLES

Introduction

An array is a set of several elements which are named by a common identifier. To further specify an individual element, a subscript is specified after the identifier.

When a fixed number of past values (history) of a variable need to be kept (as in many control algorithms), a delay variable can be used. Several applications of delay variables have already been described in chapter 1.

8.1 Subscripted Variables

General Description

A subscript is used to specify an individual element in an array.

Precise Description

A subscripted variable refers to a particular value in an integer, bit, real, or character array. See chapter 2 for a description of how an array's dimensions and type are declared. The syntax of a subscripted variable is as follows:

```
name(sub1,sub2,...,subn)
```

Where:
"name" is the identifier which refers to the array as a whole.
"sub1,sub2,...,subn" is the list of subscripts which refer to the specific element in the array. The "sub"s must be in one-to-one correspondence with the "dim"s specified when the array was declared. If the number does not agree, an error will occur. The "sub"s may be any integer or bit constant, variable, expression, or value returned by a function. If the value is real, it will be converted to integer using a truncating conversion.

A subscripted variable is treated like almost any other scalar variable. Exceptions are that it may not specify a <time>, priority, delay element, task identifier, or field description. A temporary assignment to a scalar variable can be used in these cases.

Examples:

```
TRACK = LAYOUT(PHOTOCELL_NUMBER);
POSITION(SAMPLE_NUMBER) = SAMPLED_VALUE;
IF MATRIX33(I,1) = 0 THEN CALL PIVOT;
```

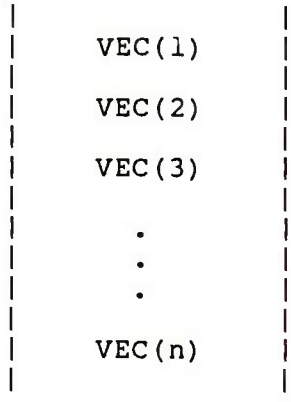
8.2 Using Lists

General Description

A list or vector is an array of singly-subscripted variables.

Precise Description

A list has just one subscript which specifies the position on the list:



Examples:

Consider the following problem: A subroutine is to be written which will compare an input string with several constant strings and return an integer specifying which string matched, or zero if no match was found. One possible way of doing this is a sequence of "IF" statements:

```
IF INBUF = 'PANIC' THEN RETURN (1);
IF INBUF = 'LIGHTS' THEN RETURN (2);

. . .

IF INBUF = 'START' THEN RETURN (9);
ELSE RETURN (0);
```

This method is awkward at best, and will require a lot of storage if many commands are to be implemented.

A more elegant method of writing the program is possible using a list. This method will produce much less object code.

```
COM_FIND: PROCEDURE(INBUF);
/* SUBROUTINE TO FIND COMMAND */

   CHARACTER(*) INBUF;           /* THE COMMAND THAT WAS INPUT */
   CHARACTER(6) COMMANDS(1:3) STATIC
      INITIAL('PANIC','LIGHTS','START');
   INTEGER INDEX;

/* SEARCH LIST OF COMMANDS */

   DO INDEX = 1 TO 3;
      IF COMMANDS(INDEX) = INBUF THEN
         RETURN INDEX;
   END;

/* COMMAND NOT IN LIST */

   RETURN (0);
END COM_FIND;
```

As a final example of the use of lists, suppose that several hundred values of data are taken. A one dimensional array of the values is kept. It is desired that the smallest and largest values be found, and the mean computed. Using a list this is easily programmed and is fairly efficient.

```
PROCESS_DATA: PROCEDURE(MEAN,SMALL,LARGE,DATA,NUM);

/*  FIND  SMALLEST  VALUE,  LARGEST  VALUE AND MEAN OF VALUES IN
"DATA" */

   REAL (MEAN,SMALL,LARGE);
   REAL DATA(*);
   REAL SUM;
   INTEGER (NUM,I);

/* INITIALIZE THE VARIABLES */

   SMALL = (10**38);
   LARGE = -(10**38);
   SUM = 0.0;

/* NOW PROCESS THE DATA ONE AT A TIME */

   DO I = 1 TO NUM;
      IF DATA(I) < SMALL THEN
         SMALL = DATA(I);
      IF DATA(I) > LARGE THEN
         LARGE = DATA(I);
      SUM = SUM + DATA(I);
   END;
   MEAN = SUM / NUM;
END PROCESS_DATA;
```

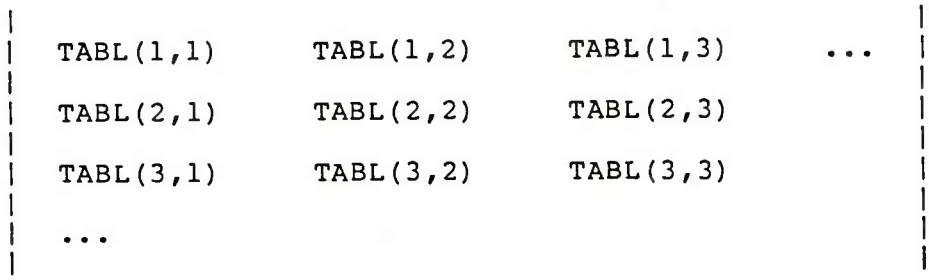
8.3 Tables, Matricies, And Multiple Subscripts

General Description

A doubly subscripted array is called a table or matrix. When needed, three or more subscripts may be used.

Precise Description

With a table, the first subscript specifies the row, and the second subscript specifies the column:



Examples:

The following program will print the contents of a table in a format like the matrix above.

```
PRINT_MATRIX: PROCEDURE(ROWS,COLUMNS,MATRIX);

/* PRINT THE CONTENTS OF A MATRIX DIMENSIONED AS: (1:ROWS,
1:COLUMNS) */

   INTEGER (ROWS,COLUMNS,I,J);
   REAL MATRIX(*,*);
   CHARACTER(120) BUFFER;

   DO I = 1 TO ROWS; /* FOR EACH ROW */
      BUFFER = '|    ';
      DO J = 1 TO COLUMNS;
         BUFFER = BUFFER || MATRIX(I,J) || '    ';
      END;
      BUFFER = BUFFER || '|';
      OUTPUT = BUFFER;
   END;

END PRINT_MATRIX;
```

8.4 Matrix Operations

General Description

Matrix operations are not yet supported by the CRASH language, but a set of subroutines are available. See chapter 11.

8.5 Delay Variables

General Description

A delay variable is a means of creating a circular list data structure. Delay variables provide a means to refer to previous values of a variable. This is necessary when, for example, the derivative with respect to time is needed to calculate the control to be applied to a system.

Precise Description

Delay variables must have the ANALOG or DISCRETE attribute specified in their declarations. The DELAY attribute specifies the number of past values to be kept. See chapter 2 for more details on declaring delay variables.

A. Referencing a Delay variable

A reference to a delay variable can be made in two ways. If just the name of the variable appears, the current (most recently assigned) value is used. If a previous value is to be referenced, the following form should be used:

```
delayvar @ amount
```

Where:
"delayvar" is the name of the delay variable.
"amount" is an integer constant or variable which specifies the amount of delay from the current value. "@0" is the current element, "@1" is the previous value.

B. Assignment to a delay variable

A circular list structure is created for each delay variable. The most recently assigned value is the current head of the list. Values can be assigned using an assignment statement (chapter 7) or the GET statement (chapter 7).

C. Using a Delay Variable for Data Acquisition and buffering

In addition to the circular list structure, a delay variable can be used to efficiently block data items into a larger record for data acquisition applications. Whenever a delay variable is created and after a PUT RECORD the variable is "empty". The number of values the variable can hold is specified with the

DELAY attribute in the declarations. After "delay" assignments or GET's to the variable it will be "full" (i.e. the next assignment to the variable will wipe out the oldest value in the circular list). An ON-condition is provided which will be triggered when this wraparound point is reached. The syntax is:

ON DELAYFULL(var1,var2,...,varn) basic statement

Where:
"var1, var2,...,varn" are delay variables. Whenever one of the variables "fills" the basic statement will be executed, and then the processor will return to whatever it was doing before the condition occured. Typically, the ON action would be to empty the variable with a PUT RECORD statement.

Examples:

The following program does really nothing of use, but illustrates the various ways of referencing delay variables.

```
DELAY_IT: PROCEDURE MAIN;
/* DELAY LINE PROGRAM */
/* VALUES WHICH ARE READ FROM A DEVICE APPEAR */
/* ON THE CONSOLE AS OUTPUT EIGHT SAMPLES LATER */

   INTEGER DELAY_LINE ANALOG DELAY(8) LDN(1);
   INTEGER I;

/* INITIALIZE THE DELAY LINE */

   DO I = 0 TO 7;
      DELAY_LINE@I = 0;
   END;

/* THIS ALSO INITIALIZES THE DELAY LINE */

   DO I = 0 TO 7;
      DELAY_LINE = 0;
   END;

/* NOW START DELAYING OUTPUT */

   DO WHILE 1; /* DO IT FOREVER */
      GET DELAY_LINE;
      OUTPUT = DELAY_LINE@7;
   END;

END DELAY_IT;
```

The DELAYFULL condition is used in the following program to read data values and format them into 80 word records.

```
DATA_AK: PROCEDURE TASK;
/* Read 1024 samples. Format in 80 word buffer and output to
floppy disk file attached to unit 0 */
   INTEGER SAMP DELAY(80) STATIC LDN(0);
   ON DELAY_FULL(SAMP) PUT RECORD(0) SAMP;
   INTEGER COUNT;
   DO COUNT = 1 TO 1024;
      GET SAMP;
   END;
END DATA_AK;
```

8.6 Subscript And Delay Checking

General Description

If desired, CRASH can perform run-time checking on the subscripts of an array or the delay specification of a delay variable whenever it is referenced. Normally, no checking is performed, but checking can be turned on with the CHECK statement.

Precise Description

The subscripts of an array must lie in the range specified or assumed (by default) in the declarations for the array. If a subscript is not in this range, an element may be referenced which is not part of the array, or an incorrect element may be referenced. If this occurs during an assignment, the program may self destruct by changing itself inadvertently.

Likewise, the delay specified with a delay variable must not be greater than the number of past values to be kept (specified in the declarations for the delay variable).

To warn the user of this occurrence, CRASH allows subscript or delay checking to be automatically performed on selected variables in an efficient manner at run-time. If an error occurs, a default value of zero or the null string is used for the variable (depending on type), and a specific statement is executed (as described below).

A. CHECK

The CHECK statement is used to turn on subscript or delay checking for an array or delay variable in the compiler. The forms are:

```
CHECK SUBSCRIPTRANGE(var,var,var,...);

CHECK DELAYRANGE(var,var,var,...);
```

Where:
"var" is the name of the array or delay variable to be CHECKed.

When a CHECK statement is encountered, code is emitted to automatically use the default value when a reference is made out of bounds. If no ON condition is set up for the variable at run-time, CRASH automatically prints a warning message when a bad reference is detected. The CHECK statement is not an executable statement, but is a switch for the compiler.

B. IGNORE

The IGNORE statement is used to turn off subscript or delay checking if it is no longer needed. The forms are:

```
IGNORE SUBSCRIPTRANGE(var,var,var,...);

IGNORE DELAYRANGE(var,var,var,...);
```

Where:
"var" is the name of the array or delay variable which no longer needs to be CHECKed.

When an IGNORE statement is processed, code is no longer emitted for the specified variables to check array and delay specifications. This does not affect the run-time checking of variables which were previously CHECKed. The IGNORE statement is not an executable statement.

C. The ON-Condition

The ON condition has been previously introduced in chapter 5. This section describes its use in specifying what action is to be taken when a DELAYRANGE or SUBSCRIPTRANGE condition occurs. The forms are:

```
ON SUBSCRIPTRANGE(var) action;

ON DELAYRANGE(var) action;
```

Where:
"var" specifies the array or delay variable which this ON statement pertains to.
"action" is one of the following:
1) START taskname
2) basic statement

The use and invocation of tasks has already been described in chapter 5.

"basic statement" is any single CRASH statement except an "IF" statement. If more than one statement must be performed, or an IF statement must be used, they can be enclosed by DO; ... END; to make it a basic statement.

When the ON statement is executed, it sets up the statement to be executed when a reference out of bounds occurs with the

variable. When this happens, the basic statement is executed, and execution resumes at the point of interruption.

D. The REVERT Condition

The REVERT statement is used to cancel the execution of the statement set up by the ON condition. The syntax is:

```
REVERT SUBSCRIPTRANGE(var,var,var,...);

REVERT DELAYRANGE(var,var,var,...);
```

Where:
"var" is a delay variable or array. The execution of a REVERT statement cancels the action set up by the ON condition but the default value will still be used in any out of range references which were under the influence of a CHECK statement at compile time. The REVERT statement only controls run-time actions.

Examples:

```
INTEGER BIG_ARRAY(1:500);                    /* DECLARE THE ARRAY */
. . .                                        /* LEAVE OUT DETAILS */
ON SUBSCRIPTRANGE(BIG_ARRAY)                    /*  WHAT TO DO ON */
   OUTPUT = 'BAD SUBSCRIPT';          /*  SUBSCRIPT OUT OF RANGE  */
. . .                                          /* MORE DETAILS */
CHECK SUBSCRIPTRANGE(BIG_ARRAY);                 /* TURN IT ON */
OUTPUT = BIG_ARRAY(I);                      /* PRINT AN ELEMENT */
IGNORE SUBSCRIPTRANGE(BIG_ARRAY);              /* TURN IT OFF */
. . .                                         /* CONTINUE ... */
```

If the value of "I" in the assignment statement is less than zero or greater than 500, the message BAD SUBSCRIPT is printed on the console.

CHAPTER 9 - CRASH ON MTS

9.1 How To Run The CRASH Compiler

General Description

At the present time, using the CRASH compiler is a three phase process. The compiler itself produces LSI-11 Assembly code. This must in turn be assembled by the LSI-11 assembler available on MTS. Finally, all object modules thus produced must be linked together with the CRASH library to produce an absolute load file which can be loaded on the LSI-11.

Precise Description

A. The Compilation Phase

The CRASH compiler is invoked by running the file K2AT:CRASH. The source code to the compiler is read from SCARDS. If enabled (see section 9.2 on Control Toggles below) a listing of the source code, variable cross-reference, and execution statistics are written on SPRINT. The assembly code is punched on SPUNCH.

Example:

```
RUN K2AT:CRASH SCARDS=MYPROG  SPUNCH=LSI11CODE  SPRINT=MYLIST
T=3
```

B. The Assembly Phase

If no errors of a severe nature were detected in the first phase, the LSI-11 code can be assembled. The *11ASR assembler is available on MTS. It reads the source code produced by CRASH through SCARDS. The object code produced is written on SPUNCH. If specified, a listing is produced on SPRINT, most CRASH users will not need this listing, however.

Example:

```
RUN *11ASR SCARDS=LSI11CODE SPUNCH=OBJ.CODE T=3
```

C. The Linking Phase

The final step is to link together all of the external procedures or modules and the necessary routines from the CRASH library. The *LINK11 program available on MTS produces an absolute load file which can be loaded on the LSI-11. There are several commands used. For a complete description, CC Memo 286 should be consulted.

See The PDP-11 Assembler and Link-Editor: A User's Guide to *11ASR and *LINK11, University of Michigan Computing Center Memo 286, 1973

Example:

```
#RUN *LINK11              Invoke linker
:SET @,0100               Starting address in hex.
:LINK OBJ.CODE            Link all
:LINK OBJ.CODE2           the external
     .
     .
     .
:LINK OBJ.CODEN           procedures.
:LINK K2AT:CRASHLIB       Link the library routines.
:WRITE MYLOAD             Punch the load file.
:STOP
```

D. The Combined Compiler and Assembler

A program is available which invokes the compiler, and if no errors were detected, then invokes the assembler. The object modules are then added to or replaced in a specified file. If a particular external procedure (module) already exists in the file, it is replaced by the new version. If the module does not exist it is added to the file. This feature of the driver program greatly eases the handling of object code when separate compilations are being performed, and thereby reduces the amount of funds needed during program development (if used properly). A provision is also made to allow the object code file and the CRASH library to be linked.

To use the driver program the following MTS command may be issued:

```
RUN K2AT:C SCARDS=MYPROG SPUNCH=OBJ.CODE SPRINT=MYLIST PAR=
```

The source code is read from SCARDS. The file attached to SPUNCH should be either empty, or contain CRASH object modules. New object modules will be added to the file; old ones will be replaced. The CRASH listing will be produced on SPRINT. If SCARDS, SPRINT, or SPUNCH is unassigned, then the user will be prompted for the files to be used (except as noted below). If a carriage return is simply entered when the user is being prompted for a file name, the infinite wastebasket *DUMMY* is assumed. If errors or severe errors are detected during the compilation phase, the assembler will not be invoked (except as noted below).

Several parameters may be specified in the PAR field to modify the operation of the driver. The valid parameters are:

D or DEFAULT --- Use default files -CRASH.S, -CRASH.L, -CRASH.P for SCARDS, SPUNCH, and SPRINT if unassigned.

ASS or ASSEMBLE --- Assemble even if errors detected.

E or EMPTY --- Empty object and listing files before using.

PE or PEMPTY --- Empty listing file before using.

OE or OEMPTY --- Empty object file before using.

L or LINK --- Link the object code with the CRASH library. The linked code will be in -LINK. An octal load map will be written to file -MAP.

CCODE --- Save emitted assembler code in file -CRASH###. Used as compiler debugging aid.

AL or ALIST --- Produce assembler listing on SPRINT.

Optionally, the parameter may be preceded by "NO" to complement the action of the parameter, however all parameters default to NO...

Example: RUN K2AT:C SCARDS=PROG SPUNCH=OBJ SPRINT=-P PAR=PE,L

9.2 Control Toggles

General Description

Whenever a dollar sign ($) is scanned within a comment, the immediate next character is taken as a control toggle character. There are 256 control toggles --- one for each EBCDIC character. Each toggle can be either "on" or "off". When the "$" is scanned, one of two things happens depending on the toggle character:

1) The corresponding toggle is complemented, if it was "on" it is turned "off", if it was "off" it is turned "on".
2) A specified action is taken.

Precise Description

The currently available control toggles are:

C --- Check Syntax Only. The source code syntax is checked by the CRASH parser. No object code is produced. Syntax checking is useful for finding missing semicolons, ENDs, primes, etc. and costs about one third the cost of running the compiler with code emission turned on. (initially off)

D --- Generate Symbolic Debugger tables. Tables are generated for each external procedure compiled with this toggle turned on, allowing the use of the real time interactive debugger as an aid in program development. The D2 toggle can be used whenever the main program tables are not needed, but other external procedures are to be debugged.

@ --- Automatic Listing Indenting. When enabled, the listing is automatically indenting to show nesting levels with vertical bars printed connecting DO;...END;'s. (initially off)

K --- Token Dump. Names of tokens and values of constants are dumped as encountered during scanning if this toggle is "on". (initially "off")

L --- A listing of the source code is produced on SPRINT if this toggle is "on". (initially "on")

N -- Listing Control. When this toggle is encountered, the immediate next character is output as carriage control before the current line is printed. Thus, a page eject can be placed in the CRASH listing by using $N1.

O -- SERCOM error printing. When enabled, error messages are echoed on sercom as detected. This toggle is initially "on" when in conversational (terminal) mode, initially "off" when in batch mode.

P --- Partial Parse Dump (initially "on")

R --- Reduction Dump. Production numbers and productions are dumped before each reduction takes place if this toggle is "on". (initially "off")

X --- Variable Cross Reference Listing. The name, internal name and references for each identifier are printed after each external procedure. (initially on).

1 --- Print expanded macros. Prints the expansion of each macro call on next line of SPRINT. (initially off);

2 --- Print macro definitions and other data on SERCOM. Debugging aid only. (initially off).

3 --- Enable Macro Expansion. Allows the scanner to recognize macro calls. Does not affect the processing of macro definitions. (initially on).

E --- List the emitted code. Lists the code for each statement after the statement on SPRINT. (initially off).

? --- Debug Breakpoint. If this toggle is "on", then every time a question mark (?) is encountered within the source code, procedure DEBUG is invoked. The general user of CRASH should not use this toggle, as it is useful only for debugging the compiler itself. For details on the use of DEBUG see file W164:EXPLDEBUG.S (for the time being anyway...)

S --- Selective Debugging. When this toggle is encountered, the user is prompted via GUSER for lists of production numbers of the form:

```
ON=prd#,prd#,prd#,...
OFF=prd#,prd#,prd#,...
ALL
NONE
```

Where prd# is either a single number (e.g. 10) or a range of numbers (e.g. 10-20)
ALL does the same thing as "ON=1-256"
NONE does the same thing as "OFF=1-256"

DEBUG will then be called before and after each reduction involving any of the productions listed.

In addition the following entries are legal:
RETURN or a null line to stop the prompting and resume compilation
$_ (where "_" can be any character). In this case the toggle corresponding to the second character in the line is complemented and its current state is printed out on SERCOM.

9.3 Including An MTS File In CRASH Source Code

General Description

The $INCLUDE command can be used to copy an MTS file into the source stream of the compiler.

Precise Description

When several external procedures use a common set of GLOBAL variables, they can be put in a separate file, and the $INCLUDE command used to include them in each procedure. Or a common set of macro definitions can be made up in a separate file if used by more than one procedure. This eliminates the need to type in identical declarations for several procedures. The $INCLUDE command appears in a comment like a control toggle and has the following form:

```
* $INCLUDE filename *
```

Where filename is the name of an MTS file to be included in the source to CRASH. This statement is identical to the MTS command $CONTINUE WITH filename RETURN appearing in column one except that the $INCLUDE statement appears on the listing to document the existence of the copied file.

Examples:

```
/* $INCLUDE K2AT:CRASHMACLIB */
/* $INCLUDE GLOBAL.DEFS */
```

CHAPTER 10 - MACROS

Introduction

A macro is a line of text which is set aside during macro definition and reintroduced into the input stream of the compiler, possibly with modifications, by a macro call and subsequent macro expansion. The macro definition is modified by the optional use of an argument list with the macro call. All macros must be defined before they are used, and CRASH allows up to 100 macros to be defined. Macros are used internally by CRASH to implement the MAP attribute.

10.1 Macro Definitions

General Description

A macro must be defined before it can be used. This is usually done along with the declarations for the procedure. Once a macro has been named and defined, any occurrence of that name will cause the macro processor to be invoked, and the text expanded (if the $3 toggle is enabled).

Precise Description

Macro definitions have the following syntax, and may occur anywhere an identifier can occur.

```
MACRO name; text; MEND;

     OR

MACRO name(par1,par2,...,parn); text; MEND;
```

Where name is any identifier, and text is any string not containing "MEND;". The text may not be longer than 255 characters.

Examples:

A simple text substitution macro definition merely substitutes the text for the name. This macro definition will cause every occurrence of the identifier "FOREVER" to be replaced by "WHILE 1".

Example:

```
MACRO FOREVER;WHILE 1;MEND;
```

A more complicated macro definition allows argument substitution in the text of the macro definition. The next section covers the passing of arguments in a macro call. The macro definition below will cause the identifier "DECLARE" to be replaced by "INTEGER zzzzz" where zzzzz is the argument passed by the macro call.

Example:

```
MACRO DECLARE(VARIABLE);INTEGER VARIABLE;MEND;
```

The macro expansions can be turned on or off as below. The control toggle $3 can also be used(Chapter 9), but the example below shows the preferred method, since the state of things is positively known with these two statements.

Example:

```
MACRO ON        or      MACRO OFF
```

10.2 Macro Calls And Text Expansion

General Description

A macro is invoked when the CRASH parser encounters an identifier which has been defined by a previous "MACRO" statement.

Precise Description

The syntax of a macro call is as follows:

```
name

  OR

name(arg1,arg2,...,argn)
```

The first type of call is invoked when just the macro name appears in the input stream of the compiler. In this case, the definition is simply copied into the input stream in place of the name. If the optional argument list is included it is required that the number of arguments specified in the argument list be in one-to-one correspondence with the list of parameters specified when the macro was defined. If too many arguments are specified, a warning message is generated, and the extra ones are ignored. For each occurrence of a parameter in the macro definition, the corresponding argument is inserted instead. This enables modification of the macro definition.

Examples:

A typical use for macros is to define an "infinite" DO- loop. In the declarations of the program, the macro definition in example one must appear. Later we could use the following statements to form the infinite loop.

Example:

```
DO FOREVER;                 /* Expands into DO WHILE 1; */
   . . .                    /* Leave out the details */
END;                        /* End of the infinite loop */
```

The Argument list enables one macro to do the work of several.

Example:

```
MACRO ARITH(PAR1,PAR2); BIGNUM = PAR1 + PAR2 ; ;MEND;

. . .     Later in the program ...
ARITH(LITTLE,MEDIUM + SMALL)     /* EXPANDS INTO */
     /*    BIGNUM = LITTLE + MEDIUM + SMALL ;     */
```

CHAPTER 11 - PREDEFINED FUNCTIONS AND SUBROUTINES

Introduction

The CRASH compiler has a library of functions and routines which are pre-defined. All a user needs to do to use one of these routines is to make a reference to it, it does not need to be declared. These routines are useful in scientific calculations, matrix operations, character string handling, and type conversions.

11.1 Mathematical Functions

General Description

CRASH has several functions predefined for evaluating the sine, cosine, arc-tangent, logarithm, natural anti-logarithm, square root, and for generating uniform random numbers. All of these functions take a real number argument and return a real result.

Precise Description

A. ATAN(x)

The ATAN function is invoked with one real parameter, and returns the arc-tangent in radians.

Example:

```
BEARING = ATAN(X + Y + Z);
```

B. COS(x)

The COS function is called with an angle specified in radians. The cosine is returned.

Example:

```
PHASE = COS(2*ANGLE);
```

C. EXP(x)

The EXP function returns the natural anti-logarithm of the argument (e**x).

Example:

```
Y(T) = EXP(X(T));
```

D. LOG(x)

The LOG function is invoked with a positive real number parameter and returns the natural logarithm of the number.

Example:

```
X_COORDINATE = LOG(A * 3.67);
```

E. SIN(x)

The SIN function returns the sine of an angle specified in radians.

Example:

```
V(T) = SIN(THETA);
```

F. SQRT(x)

The SQRT function returns the square root of the real parameter. If the value is negative, the absolute value is used, and a negative result returned.

Example:

```
R = SQRT(X**2 + Y**2);
```

G. URAND

Function URAND computes a uniformly distributed random number sequence from a real number seed. Each time URAND is called, a real number value between 0 and 1 is returned. In addition, the seed is changed according to the random number generating algorithm. To generate a sequence of random numbers, then, all that is necessary is to initialize the seed to some arbitrary value. each call to URAND will update the seed automatically.

Example:

```
RAND = URAND(SEED);
```

11.2 Inline Functions

General Description

Some of the pre-defined functions are one or two instructions. Since they are so trivial, CRASH emits very efficient inline code to perform these functions.

Precise Description

A. LENGTH(x)

The LENGTH function returns the current length of the character parameter.

Example:

```
BUFLEN = LENGTH(BUFFER);
```

B. MAXLEN(x)

The MAXLEN function returns the maximum number of characters that will fit in the space allocated to the parameter.

Example:

```
SIZE = MAXLEN(BUFFER);
```

C. ADDR(x)

The ADDR function returns the address of the argument in the LSI-11 memory. If a scalar or array element is specified, the value returned is the address of the first byte of the variable. If "x" is an array or delay variable, the dope vector address is returned. If "x" is a procedure or task, the entry address is returned. "x" may not be an expression.

Example:

```
STARTING_ADDRESS = ADDR(TASKA);
```

D. NUMARGS

A CRASH subroutine can be called with fewer parameters than were declared in its procedure definition. The NUMARGS function can be used to determine the actual number of parameters passed to the subroutine.

Example:

```
SUB1: PROCEDURE (ARG1,ARG2,ARG3);
   INTEGER ARG1,ARG2,ARG3;
   IF NUMARGS < 3 THEN SHORTCALL = TRUE;
      .
      .
      .
END SUB1;
```

E. ABS(x)

The ABS function returns the absolute value of the real or integer parameter. If a real value is passed the value returned is real, otherwise an integer is returned.

Example:

```
Z = ABS(X - 3.0);
```

11.3 Matrix Operations

General Description

Several subroutines are available in the CRASH library to perform matrix arithmetic, copying, conversions, and logical operations. All parameters to these routines are unsubscripted variable names. These routines are appreciably more efficient than writing DO loops to perform the given functions.

Precise Description

In all cases below, the C array is the destination array, and must not be the same array as A or B.

A. IMTXADD and FMTXADD

The operation C = A + B is performed by these two subroutines, where A,B, and C are arrays with the same dimensions. FMTXADD is for real arrays, and IMTXADD is for integer arrays.

Example:

```
CALL FMTXADD(A,B,C);
CALL IMTXADD(A,B,C);
```

B. IMTXMUL, BMTXMUL, and FMTXMUL

The operation C = A * B is performed. A and B must be two dimensional matricies, with the number of columns in A equal to the number of rows in B. C must have the same number of rows as A and the same number of columns as B. IMTXMUL does integer multiplication and FMTXMUL does floating point arithmetic. BMTXMUL performs Boolean multiplication, with "times" replaced by "and" and "plus" replaced by "or".

Example:

```
CALL IMTXMUL(A,B,C);
CALL FMTXMUL(A,B,C);
CALL BMTXMUL(A,B,C);
```

C. IMTXSUB and FMTXSUB

The operation C = A - B is performed, where A, B, and C are arrays with the same dimensions. IMTXSUB is for integer arrays and FMTXSUB is for real arrays.

Example:

```
CALL IMTXSUB(A,B,C);
CALL FMTXSUB(A,B,C);
```

D. ISCLDIV and FSCLDIV

The operation C = B / A is performed, where B and C are like dimensioned arrays, and A is a scalar. ISCLDIV is for integer arrays and FSCLDIV is for real arrays.

Example:

```
CALL FSCLDIV(A,B,C);
CALL ISCLDIV(A,B,C);
```

E. ISCLMUL and FSCLMUL

The operation C = A * B is performed, where B and C are like dimensioned arrays, and A is a scalar. ISCLMUL is for integer arrays and FSCLMUL is for real arrays.

Example:

```
CALL FSCLMUL(A,B,C);
CALL ISCLMUL(A,B,C);
```

F. IMTXAND

The operation C = A AND B is performed, where A, B, and C are integer or bit arrays of equal dimensions.

Example:

```
CALL IMTXAND(A,B,C);
```

G. IMTXOR

The operation C = A OR B is performed, where A, B, and C are integer or bit arrays of equal dimensions.

Example:

```
CALL IMTXOR(A,B,C);
```

H. IMTXXOR

The operation C = A XOR B is performed, where A, B, and C are integer or bit arrays of equal dimensions.

Example:

```
CALL IMTXXOR(A,B,C);
```

J. MTXMOV

The "A" matrix is copied into the "B" matrix. Both matricies must have the same dimensions. Integer, real, or character arrays can be moved by MTXMOV.

Example:

```
CALL MTXMOV(A,B);
```

K. ARRAYINFO

Subroutine ARRAYINFO can be used to determine the actual and virtual bases of an array, the number of dimensions and the elementsize (in bytes), and the lower and upper bounds for each dimension of an array. There are three calls:

```
CALL ARRAYINFO(-1,ARRAY,ACTUAL,VIRTUAL);
CALL ARRAYINFO(0,ARRAY,ELEMENTSIZE,NDIMS);
CALL ARRAYINFO(DIMNUM,ARRAY,LB,UB);
```

Where:

"ARRAY" is the array in question.

"ACTUAL" is an integer variable to store the actual starting address of the array storage area in.

"VIRTUAL" is an integer variable to store the virtual base (see CRASH run-time strategy report) in.

"ELEMENTSIZE" is an integer to store the number of bytes reserved for each element in the array.

"NDIMS" will contain the number of dimensions of the array.

"DIMNUM" is an integer (1 to NDIMS) specifying which dimension lower bound and upper bound information is wanted.

"LB" will contain the lower bound for dimension DIMNUM.

"UB" will contain the upper bound for dimension DIMNUM.

11.4 Matrix Conversions

General Description

Two subroutines are provided to allow real-to-integer and integer-to-real matrix conversions.

Precise Description

A. FMTX2I

This subroutine converts a real array to an integer array. Both arrays must have the same dimensions.

Example:

```
CALL FMTX2I(REAL_ARRAY,INTEGER_ARRAY);
```

B. IMTX2F

This subroutine converts an integer array to a real array. Both arrays must have the same dimensions.

Example:

```
CALL IMTX2F(INTEGER_ARRAY,REAL_ARRAY);
```

11.5 Character To Numerical Conversion

General Description

Included in the CRASH library are three subroutines to convert ASCII character strings into integer or real numbers.

Precise Description

A. D2BIN

D2BIN is a CRASH function which attempts character to integer conversion. The calling sequence is:

```
INT_VAR = D2BIN(STRING,BRKCHAR,IERR);
```

Where:
"STRING" is the character expression to be converted.
"BRKCHAR" is an integer variable which is modified by D2BIN to specify the number of the character which caused conversion to stop
"IERR" is an integer variable which will contain the return code.

IERR is set to one of the following return codes:

0 -- Conversion was terminated by a blank or comma
1 -- Conversion overflow
2 -- End of line was reached
3 -- A character other than a comma or blank terminated conversion
4 -- A null string was passed

Examples:

```
STRING = ' 123 ' BRKCHAR = 4 IERR = 0
STRING = '123'   BRKCHAR = 3 IERR = 2
STRING = '1234A' BRKCHAR = 4 IERR = 3
STRING = '99999' BRKCHAR = 3 IERR = 1
```

B. D2FLOAT

D2FLOAT is a CRASH callable function which will attempt to convert a character string into a floating point (REAL) number. The calling sequence is:

```
REAL_VAR = D2FLOAT(STRING,BRKCHAR,IERR);
```

Where:
"STRING" is the character variable to be converted.
"REAL_VAR" is a REAL variable which will have the value of the converted number.
"BRKCHAR" is the index which points to the character which stopped conversion.

"IERR" is the return code.

Valid return codes are:

```
0 -- Conversion normal
1 -- Illegal character in input string
2 -- Null string passed to D2FLOAT
3 -- Conversion overflow
```

Examples:

```
STRING='2.3' BRKCHAR = 3 IERR = 0
STRING='+2.3 ' BRKCHAR = 4 IERR = 0
STRING='1.0E-55' BRKCHAR = 7 IERR = 3
STRING='2.3A' BRKCHAR = 3 IERR = 1
STRING='99999.0' BRKCHAR = 7 IERR = 0
```

C. O2BIN

O2BIN is a CRASH function which attempts octal character to integer conversion. The calling sequence is:

```
INT_VAR = O2BIN(STRING,BRKCHAR,IERR);
```

Where:
"STRING" is the character expression to be converted.
"BRKCHAR" is an integer variable which is modified by O2BIN to specify the number of the character which caused conversion to stop
"IERR" is an integer variable which will contain the return code.

IERR is set to one of the following return codes:

```
0 -- Conversion was terminated by a blank or comma
1 -- Conversion overflow
2 -- End of line was reached
3 -- A character other than a comma or blank
     terminated conversion
     (an illegal character was encountered)
4 -- A null string was passed
```

Examples:

```
STRING = ' 123 '      BRKCHAR = 4 IERR = 0
STRING = '123'        BRKCHAR = 3 IERR = 2
STRING = '1234A'      BRKCHAR = 4 IERR = 3
STRING = '777777777'  BRKCHAR = 4 IERR = 1
STRING = '89'         BRKCHAR = 1 IERR = 3
```

11.6 Numerical To Character Conversion

General Description

The CRASH library also contains a subroutine to convert integers to octal format ASCII character strings.

Precise Description

A. BIN2O

BIN2O is a CRASH function which attempts integer to octal format character conversion. The calling sequence is:

```
CHAR_VAR = BIN2O(INT_VAR);
```

No errors are detected by this routine. The number is converted into octal integer format, and placed right justified in a field 8 places wide, padded on the left with blanks if necessary. Note that the destination variable in a BIN2O conversion must be 8 or more characters wide.

11.7 OSWIT Interface Routines

General Description

Several subroutines and functions allow CRASH routines to make system calls to OSWIT. Two other routines allow the user access to memory by address. A function is also provided which returns the value of the parameter field specified on the OSWIT RUN command which invoked the MAIN procedure.

Precise Description

A. SYSTEM

Subroutine SYSTEM is simply called and causes immediate termination of all active tasks including the MAIN procedure.

Example:

```
CALL SYSTEM;
```

B. OSWIT

Subroutine OSWIT is also simply called. It will call the OSWIT command handler. If the user issues a RESTART command to OSWIT, the procedure will continue executing.

C. PARFIELD

PARFIELD is a character function which returns the value of the parameter field specified on the OSWIT RUN command. If no parameter field was given, the null string is returned.

Example:

```
OPTIONS = PARFIELD;
```

D. READ

The READ subroutine allows record reads on any legal unit number. The calling sequence is:

```
IRC = READ(unitnumber,inputstring);
```

Where:
"unitnumber" is an valid integer unit number.
"inputstring" is the character variable to contain the record which is to be read. No length check is performed so the buffer must be long enough for the maximum length record which is expected.
"IRC" is an integer which will contain one of the following return codes:

```
0 - OK
1 - End of file reached.
2 - Input line greater than 255 characters.
3 - Bad unit number.
```

Routine READ performs an automatic wait for the input operation to be completed.

Examples:

```
IF READ(26,INBUF) NOT = 0 THEN CALL IOERROR;
CALL READ(0,DATABUF);
```

E. WRITE

The WRITE subroutine allows record writes on any legal unit number. The calling sequence is:

```
IRC = WRITE(unitnumber,outputstring);
```

Where:
"unitnumber" is an valid integer unit number.
"outputstring" is the character variable containing the record which is to be written.
"IRC" is an integer which will contain one of the following return codes:

```
0 - OK
```

1 - End of disk reached.
3 - Bad unit number.

Routine WRITE performs an automatic wait for the output operation to be completed.

Examples:

```
IF WRITE(28,OUTBUF) NOT = 0 THEN CALL IOERROR;
CALL WRITE(0,DATABUF);
```

F. OPEN

This function is used to assign a file or pseudo-device to a unit number. The prototype call is:

```
IRC = OPEN(STRING);
```

Where:
"STRING" is a character variable containing a legal OSWIT assignment string.
"IRC" is an integer which will contain one of the following return codes:

0 - OK
1 - File or device doesn't exist.
2 - Bad unit number.
3 - Illegal assignment string.

OPEN first closes the unit, reverting to the default of *MSOURCE* (the console). Then an attempt is made to open the unit attached to the specified device. If the OPEN fails, the unit will be left attached to *MSOURCE*.

Examples:

```
IF OPEN('0=*CONVERTER0*') NOT = 0 THEN CALL OPENERROR;
CALL OPEN('SPRINT=*DUMMY*');
```

G. CLOSE

Function CLOSE is used to close a logical unit number. The default assignment of all units is to *MSOURCE* (the console). The prototype call is:

```
IRC = CLOSE(UNITNUM);
```

Where:
"UNITNUM" is an integer unit number to be closed.
"IRC" is one of the following return codes:

0 - OK
1 - Bad unit number.

Example:

```
IF CLOSE(1) NOT = 0 THEN CALL CLOSEERROR;
```

H. SETPFX

This function is used to change the input prompt character for the console. The calling sequence is:

```
IRC = SETPFX(UNITNUM,PROMPTCHAR);
```

Where:
"UNITNUM" is the unit number to change the prefix on.
"PROMPTCHAR" is a single character prompt. The default prompt is a question mark (?).
"IRC" one of the following integer return codes:

0 - OK
1 - Bad unit number.

Example: (to change the SCARDS prompt character to a *)

```
CALL SETPFX(26,'*');
```

J. PEEK

Function PEEK returns the value of a word in memory as an integer. The calling sequence is:

```
IVAL = PEEK(ADDR);
```

Where:
"ADDR" is an address of a memory location. If odd, the value is truncated.

Example:

```
CONVERT = PEEK(O"177170");
```

K. POKE

Subroutine POKE is used to change memory locations and to write to devices by address. The call is:

```
CALL POKE(ADDRESS,VALUE,...,VALUEN);
```

Where:
"ADDRESS" is an integer memory address.
"VALUE" is one or more integer values to be stored in consecutive memory locations beginning with ADDRESS. Up to 62 values may be specified in a single call.

Examples:

```
CALL POKE(O"167752",TINFO);
CALL POKE(O"400",0,1,2,3);
```

CHAPTER 12 - WARNINGS, ERRORS, AND SEVERE ERRORS

Introduction

The CRASH compiler has three levels of errors: WARNING, ERROR, and SEVERE ERROR. If appropriate, several different attempts at error recovery will be tried, to salvage the rest of the compilation. If things get really bad, even syntax-checking is stopped, and the compiler shuts down.

12.1 Warnings

General Description

A WARNING message is generated by CRASH whenever an error is detected which should not affect the compilation. The message states what action was taken by the compiler to recover from the minor error. The user should check this to make sure it will have no effects on the desired program operation.

Precise Description

A. Constant Warnings

"CN-02. LENGTH OF STRING GREATER THAN 256 HAS BEEN TRUNCATED"

B. Declaration Warnings

"DC-22. MORE THAN ONE LDN SPECIFIED--EARLIEST SPECIFICATION WILL BE USED."
"DC-26. 'OFFSET' SPECIFICATION MUST BE REAL--CONVERSION PERFORMED"
"DC-27. 'SCALE' SPECIFICATION MUST BE REAL--CONVERSION PERFORMED."
"DC-64. INTEGER INITIAL VALUE GIVEN IN A 'REAL' DECLARATION--CONVERSION PERFORMED."
"DC-65. REAL INITIAL VALUE GIVEN IN A 'INTEGER' DECLARATION--CONVERSION PERFORMED."
"DC-66. LENGTH OF INITIAL VALUE EXCEEDS MAXIMUM LENGTH OF VARIABLE--TRUNCATION HAS OCCURED."
"DC-67. NUMBER OF INITIAL VALUES EXCEEDS ARRAY SIZE."
"DC-68. NUMBER OF INITIAL VALUES EXCEEDS DELAY SIZE."
"DC-69. NUMBER OF INITIAL VALUES EXCEEDS 1 FOR SCALAR"

C. DO Warnings

"DO-03. LABEL ON EXIT STATEMENT IS UNDEFINED--INNERMOST 'DO' ASSUMED."
"DO-04. LABEL ON EXIT STATEMENT DOES NOT MATCH AN ACTIVE DO. INNERMOST DO ASSUMED"

D. END Warnings

"EN-01. ---AN UNLABELED 'END' HAS TERMINATED PROCEDURE-"
"EN-03. ---LABEL ON 'END' DOES NOT MATCH PROCEDURE NAME: "
"EN-04. ---LABEL ON 'END' DOES NOT MATCH AN ACTIVE 'DO'.
INNERMOST 'DO' ASSUMED."

E. Macro Warnings

"MA-01. CAN MAP ONLY BIT, INTEGER, AND CHARACTER VARIABLES."
"MA-02. MACRO DEFINITION TABLE OVERFLOW, SIZE="
"MA-08. TOO MANY ARGUMENTS IN MACRO CALL "
"MA-09. MACRO PREVIOUSLY DEFINED: "

F. Procedure Warnings

"PR-03. PROCEDURE TYPE DOES NOT MATCH DECLARATION, THE DECLARED
TYPE WILL BE USED: "

G. Miscellaneous Warnings

"WA-00. PRECEEDING ERROR MAY CAUSE CRASH TO FLAG NEXT CARD BY
MISTAKE"
"WA-01. CHECKING ABORTED ON CARD"

12.2 Errors

General Description

An ERROR message is generated whenever an error is detected which won't affect the compilation of the program, but probably will affect the execution of the program.

Precise Description

A. Symbol Errors

"SY-03. MISSING EOF"

B. Undeclared Variable Errors

"UV-01. UNDEFINED SYMBOL:"
"UV-05. UNDECLARED PARAMETER. ASSUMED TYPE IS SCALAR INTEGER:"
"UV-06. UNDECLARED PROCEDURE. IT WILL BE ADDED IF POSSIBLE: "

C. Constant Errors

"CN-10. UNEXPECTED SEMICOLON HAS TERMINATED BIT STRING"
"CN-11. CHARACTER STRING TERMINATED AT SEMICOLON"
"CN-12. ILLEGAL BIT STRING LENGTH--16 ASSUMED."
"CN-13. ILLEGAL CHARACTER STRING LENGTH--255 ASSUMED."

D. Procedure Errors

"PR-04. PROCEDURE CHARACTER SIZE DOES NOT MATCH DECLARATION, THE DECLARED SIZE WILL BE USED: "

E. Variable Errors

"VA-06. LABEL IS MULTIPLY DEFINED, IT WILL BE IGNORED: "
"VA-07. DUPLICATE PARAMETER NAME, IT WILL BE IGNORED: "

F. Miscellaneous Errors

" - . "OUTPUT" ON RIGHT-HAND SIDE OF EXPRESSION."
" - . "INPUT" ON LEFT-HAND SIDE OF EXPRESSION."
" - . ILLEGAL CONVERSION."
" - . OUTPUT = OUTPUT; ??????"
" - . ATTEMPT TO INPUT TO AN ILLEGAL STORAGE TYPE."
" - . ATTEMPT TO ASSIGN TO AN ILLEGAL STORAGE TYPE."
" - . ILLEGAL TYPE IN LOGICAL OPERATION."

G. DO Errors

"DO-06. LABEL ON 'NEXT' STATEMENT IS UNDEFINED: INNERMOST
"DO-07. LABEL ON 'NEXT' DOES NOT MATCH AN ACTIVE DO. INNERMOST 'DO' ASSUMED."

H. END Errors

"EN-02. MISSING 'END' ON ONE OR MORE 'DO'S. THEY ARE SUPPLIED."

I. Declaration Errors

"DC-00. THIS VARIABLE HAS ALREADY BEEN DECLARED: "
"DC-01. ATTRIBUTE 'GLOBAL' SPECIFIED MORE THAN ONCE."
"DC-02. ATTRIBUTE 'STATIC' SPECIFIED MORE THAN ONCE."
"DC-03. ATTRIBUTE 'DISCRETE' SPECIFIED MORE THAN ONCE."
"DC-04. ATTRIBUTE 'ANALOG' SPECIFIED MORE THAN ONCE."
"DC-06. GLOBAL VARIABLES CANNOT HAVE 'STATIC' ATTRIBUTE-- STATIC IGNORED."
"DC-07. GLOBAL VARIABLES CANNOT HAVE INITIAL ATTRIBUTE UNLESS IN MAIN PROCEDURE--INITIAL IGNORED."
"DC-10. ARRAY VARIABLES CANNOT HAVE 'DISCRETE' ATTRIBUTE-- DISCRETE IGNORED."
"DC-11. DYNAMIC ARRAYS CANNOT HAVE 'INITIAL' ATTRIBUTE--INITIAL IGNORED."
"DC-12. A '*' SHOULD BE USED HERE IN THE PARAMETER LIST-- PASSED SIZES WILL BE USED."
"DC-13. A '*' FIELD IS MEANINGFUL ONLY FOR PARAMETERS."
"DC-20. STATIC VARIABLES CANNOT HAVE 'GLOBAL' ATTRIBUTE-- GLOBAL IGNORED."
"DC-24. MORE THAN ONE DELAY SPECIFIED--EARLIEST SPECIFICATION WILL BE USED."
"DC-25. MORE THAN ONE OFFSET SPECIFIED--EARLIEST SPECIFICATION WILL BE USED."

"DC-30. PROCEDURES CANNOT HAVE 'GLOBAL' ATTRIBUTE-- GLOBAL IGNORED."
"DC-31. PROCEDURES CANNOT HAVE 'INITIAL' ATTRIBUTE-- INITIAL IGNORED."
"DC-32. PROCEDURE CANNOT HAVE 'STATIC' ATTRIBUTE-- STATIC IGNORED."
"DC-33. PROCEDURES CANNOT HAVE 'INITIAL' ATTRIBUTE-- INITIAL IGNORED."
"DC-34. PROCEDURES CANNOT HAVE 'DISCRETE' ATTRIBUTE-- DISCRETE IGNORED."
"DC-40. 'ANALOG' VARIABLES CANNOT HAVE DISCRETE ATTRIBUTE-- DISCRETE IGNORED."
"DC-50. 'DISCRETE' VARIABLES CANNOT HAVE 'ANALOG' ATTRIBUTE-- ANALOG IGNORED."
"DC-51. 'DISCRETE' VARIABLES CANNOT HAVE 'OFFSET' ATTRIBUTE-- OFFSET IGNORED."
"DC-52. 'DISCRETE' VARIABLES CANNOT HAVE 'SCALE' ATTRIBUTE-- SCALE IGNORED."
"DC-61. INITIAL VALUES MUST BE INTEGER FOR INTEGER VARIABLES."
"DC-62. INITIAL VALUES MUST BE CHARACTER FOR CHARACTER VARIABLES."
"DC-63. INITIAL VALUES MUST BE REAL FOR REAL VARIABLES."
"DC-70. MORE THAN ONE INITIAL SPECIFIED--EARLIEST SPECIFICATION WILL BE USED."

J. Macro Errors

"MA-03. ILLEGAL CHARACTER IN MACRO "
"MA-05. ILLEGAL PARAMETER FIELD DELIMITER "
"MA-06. MISMATCHED PARENTHESIS IN MACRO "
"MA-07. IMPROPER MACRO TEXT DELIMITER "

12.3 Severe Errors

General Description

When a SEVERE ERROR is detected, the compiler has found an error which will hopelessly mess up the object produced. Hence, assembly code generation is stopped. To facilitate the error finding process, syntax checking continues normally. If more than 10 severe errors are detected, the compilation is terminated. SEVERE ERRORS include syntax errors - an illegal form of a statement and illegal characters. In this case, several attempts are made to flush the current statement until a fresh start can be made with a new statement.

Precise Description

A. Symbol severe errors

"SY-01. ILLEGAL SYMBOL PAIR: "
"SY-02. NO PRODUCTION IS APPLICABLE ."
"SY-04. UNEXPECTED EOF"
"SY-05. EOF AT INVALID POINT. "

B. Constant Severe Errors

"CN-00. INVALID TYPE FOR CONSTANT."
"CN-01. ILLEGAL CHARACTER IN BIT STRING:"

C. Variable Severe Errors

"VA-00. ILLEGAL CHARACTER: "
"VA-01. ILLEGAL FIRST CHARACTER FOR IDENTIFIER:"
"VA-03. VARIABLE NOT DECLARED AN ARRAY: "
"VA-04. THIS VARIABLE WAS NOT DECLARED AS AN INTERNAL PROCEDURE: "
"VA-05. THIS PROCEDURE HAS ALREADY BEEN DEFINED: "
"VA-09. RESERVED WORD USED IN ARITHMETIC EXPRESSION: "
"VA-20. ONLY INTEGERS ALLOWED WITH MOD FUNCTION"
"VA-21. CHARACTER VARIABLE USED IN ARITHMETIC EXPRESSION: "
"VA-30. INCORRECT NUMBER OF SUBSCRIPTS SPECIFIED FOR ARRAY."

D. Compiler Severe Errors

"CO-08. STACK OVERFLOW"
"CO-42. DO-STACK OVERFLOW (TOO MANY DO'S)"

E. Undefined Variable Severe Errors

"UV-01. UNDECLARED PARAMETER USED IN ARITHMETIC EXPRESSION: "
"UV-02. TOO MANY UNDEFINED VARIABLES IN PROCEDURE (10 IS MAXIMUM)"

F. Condition Severe Errors

"CH-01. ILLEGAL CONTEXT FOR CONDITION: "
"CH-02. ATTEMPT TO SUBSCRIPTRANGE CHECK SOMETHING WHICH ISN'T AN ARRAY: "
"CH-03. ATTEMPT TO STRINGRANGE CHECK SOMETHING WHICH ISN'T A CHARACTER STRING: "
"CH-04. ATTEMPT TO DELAYOVERFLOW CHECK SOMETHING WHICH ISN'T A DELAY VARIABLE: "

G. Procedure Severe Errors

"PR-01. ATTEMPT TO CALL SOMETHING WHICH ISN'T A PROCEDURE: "
"PR-02. PROCEDURE NAME NOT A LEGAL ARGUMENT: "
"PR-04. NESTING LEVEL IS TOO DEEP FOR PROCEDURE: "
"PR-05. PROCEDURE NAME USED AS VARIABLE: "

H. Task Severe Errors

"TA-01. ONLY ONE TASK NUMBER MAY BE SPECIFIED FOR "
"TA-02. NO TASK IDENTIFIER SPECIFIED FOR "
"TA-03. TASK CANNOT BE SCHEDULED WITHOUT PRIORITY SPECIFICATION."

I. DO Severe Errors

"DO-01. ITERATION VARIABLE NOT AN INTEGER"
"DO-02. ATTEMPT TO EXIT A NON-EXISTENT DO. STATEMENT IGNORED."
"DO-05. ATTEMPT TO 'NEXT' A NON-EXISTENT DO.--STATEMENT
IGNORED."

J. Declaration Severe Errors

"DC-14. DIMENSION LIST MUST APPEAR BEFORE ANY ATTRIBUTES."
"DC-15. DIMENSIONS MUST BE INTEGER VALUES."
"DC-16. LOWER BOUND EXCEEDS UPPER BOUND."
"DC-21. ILLEGAL LDN SPECIFIED--NO ASSIGNMENT WILL BE MADE."
"DC-22. ILLEGAL DELAY SPECIFIED--A DELAY OF 1 IS ASSIGNED."
"DC-35. ILLEGAL PROCEDURE DEFINITION."
"DC-36. PROCEDURE CANNOT HAVE A DIMENSION LIST."
"DC-53. 'DISCRETE' VARIABLES MUST BE INTEGER."
"DC-60. 'INITIAL' MUST BE LAST ATTRIBUTE."
"DC-80. ILLEGAL ATTRIBUTE FOR A GIVEN TYPE."
"DC-81. ILLEGAL ATTRIBUTE FOR AN ARRAY DECLARATION."
"DC-82. ILLEGAL ATTRIBUTE FOR A PROCEDURE DECLARATION."
"DC-83. ILLEGAL DECLARATION FOR A PARAMETER."

K. Bit Selection Severe Errors

"BS-01. FIELD DESCRIPTOR MUST BE INTEGER VARIABLE OR CONSTANT."
"BS-02. FIELD DESCRIPTOR IS [FIRST,LENGTH] ."

CHAPTER 13 - KNOWN COMPILER BUGS AND RESTRICTIONS

1. Character arrays cannot have a string length of 255.

2. Assignments to DELAY variables from INPUT may not contain a delay indicator (e.g. D2=INPUT is not allowed). Due to improper implementation, any delay indicator specified is ignored.

CHAPTER 14 - RAID SYMBOLIC DEBUGGER

Introduction

RAID, or Real-Time Applications Interactive Debugger is an optionally invoked programming aid which can be used to control the execution of a user CRASH program. Facilities are provided for monitoring the contents of program variables at user specified breakpoints in external procedures. With its complement of commands, RAID can help get the bugs out of user programs.

14.1 General Information

RAID will be invoked upon program execution if either parameter /* $D */ or /* $D2 */ precedes the CRASH main procedure definition in the source file used at compilation. Additionally, each external procedure compiled separately from the main must have one of the parameters precede its procedure definition.

These parameters cause symbol table information to be emitted along with the CRASH object, allowing RAID access to variables, statements, and procedures. /* $D */ outputs tables for the main procedure and all external procedures compiled with the main. For large programs, the extra memory requirements of the tables may exceed the LSI-11 memory capacity. To shrink table size the /* $D2 */ parameter omits tables for the main procedure but includes tables for inter procedure tracing. Thus this option is for the user who does not require RAID to debug the main procedure.

RAID resides in K2AT:CRASHLIB and must be linked using *LINK11 with user's object.

14.2 Statement Numbers

To the left of each line in a CRASH listing appears a statement number, each external procedure having its own set of them. Each executable statement within an external procedure is assigned its own statement number. Nonexecutable CRASH statements (such as declarations) also have statement numbers; however, that same statement number will also be assigned to the first executable statement which follows.

Because CRASH is a free format language, several statements may appear on the same line, yet only one statement number precedes the whole line. The printed statement number is assigned to the first executable statement on the line. The next consecutive integer is assigned as the statement number for the next executable CRASH statement on the line, and so on. The statement numbers on subsequent lines are adjusted to account for multi statements on preceding lines.

Statement numbers are used in RAID as a means of setting breakpoints, places where program execution can be halted. Additionally, statement numbers are used as a means of indicating where execution is halted when stepping through the program with the STEP command. In both cases, halting at a statement occurs before the statement has been executed.

14.3 RAID Mode

General Description

RAID is entered upon the initial execution of the user program, upon termination of a STEP command, and upon execution of a breakpoint. RAID consists of a set of commands which allow the user to monitor variables, and decide other stopping points in the user program where RAID is entered.

The RAID commands are entered by keyboard in any truncated form and are recognized when enough of the command name is inputted to distinguish it from other commands. Any required parameters should appear on the same line as the command, separated by blanks or commas.

The following describes each command; the full command name is followed by any required parameters denoted in brackets.

Precise Description

A. PROC

The PROC command specifies the external procedure in which breakpoints are to be set or restored. Upon entry, the default value is the main procedure. The full name of the external procedure should be entered as the parameter. The command format is:

```
PROC <external procedure name>
```

Examples:

```
PROC   MAINP
P   TASK1        (abbreviated form)
```

B. BREAK

The BREAK command is used to set breakpoints at statement numbers in the external procedure specified by the PROC command. Optionally, an iteration counts can be specified with the statement numbers. The command format is:

```
BREAK <statement_number> [@<iteration_count>,...,
<statement_number>@<iteration_count>]
```

RAID accommodates up to 16 breakpoints. Breakpoints are only set at the executable statements within the external procedure specified by the PROC command and several breakpoints may be set with a single command. When the user program reaches a breakpoint, RAID is entered.

An optional iteration count may be specified with the statement number. This allows the statement to be executed a number of times without a break and then return control to RAID on the nth time, where n is the iteration count. This is useful when breakpoints are set in a loop, and the user does not wish to halt execution on every iteration through the loop. The maximum <iteration_count> is 32,767 and the count must be greater than zero. <iteration_count> is specified for a breakpoint by appending an @ sign followed by the <iteration_count> (with no intervening blanks) to the statement number. If no @ or <iteration_count> appears, a default value of 1 is assumed. <iteration_count>s may be changed by just entering the statement number with new <iteration_count> according to the previously described format.

Examples:

```
BREAK   3, 5@2, 6@7
BR   4, 6, 10
BRE   7
```

C. RESTORE

The RESTORE command removes breakpoints from statements in the external procedure specified by the PROC COMMAND. The command format is:

```
RESTORE <statement_number> [,...,<statement_number>]
```

Specifying the statement number will remove the that statement. Note that this can even be done to remove the current breakpoint.

Examples:

```
RE   5,7,9
RESTORE   10
```

D. CLEAN

The CLEAN command restores all breakpoints set in all procedures. The command format is:

```
CLEAN
```

Examples

```
CL
CLEAN
```

E. LIST

The LIST command is used to display all breakpoints currently set . The command format is:

```
LIST
```

For each breakpoint set, a message is outputted stating the statement number, external procedure, and <iteration_count> for that breakpoint. If no breaks are set, nothing is printed.

Examples:

```
LI
LIST
```

F. RUN

The RUN command initiates execution of the user program under RAID, and may be used to continue user program from a breakpoint or step point.

Examples:

```
RU
RUN
```

G. CONTINUE

The CONTINUE command performs the same function as the RUN command.

Examples:

```
CO
CONT
CONTINUE
```

H. STEP

The STEP command causes execution of the <number of statements> specified before returning control to RAID. If no parameter is given, a default of 1 is assumed. The command format is:

```
STEP <number_of_statements>
```

The STEP command has some unique aspects. Statements executed are only counted against the step command when they are in the same internal procedure as the one the STEP was given in. Thus a CALL to another procedure and all the statements executed as a result of that call count as only one statement.

If a STEP command is given in an internal procedure which would cause the procedure to return, the STEP is terminated before the return and control goes to RAID.

Should a breakpoint be encountered before the STEP is completed, the step will resume execution if a RUN or CONTINUE is issued from the breakpoint. However, if a new STEP command is given at the breakpoint, the old STEP command is lost.

Examples:

```
ST   1
STEP   5
```

Disclaimer: There is a hardware bug in the LSI-11 which may cause RAID to go into on infinite loop while tracing with a STEP command through a multiply operation. Should this happen, press the break button on the DECwriter, followed by a P. Doing this several times may bring it out of the loop. Note that multiply instructions may be executed in the following types of statements: array subscripting, multiplications, character to numeric type conversions, and scheduling statements.

J. OSWIT

The OSWIT command calls the operating system. To return, just issue the RESTART command in OSWIT.

Examples:

```
O
OSWIT
```

K. EXIT

The EXIT command terminates the user program and unloads it.

Examples:

```
E
EXIT
```

L. LOCK

The LOCK command prevents tasks from starting while under control of RAID The priority while in RAID is raised to the highest level when the command is given, and any time a breakpoint or step returns control to RAID. STEP or CONTINUE then reduce the priority while the user program executes.

Examples:

```
LO
LOCK
```

M. UNLOCK

The UNLOCK command allows tasks to execute while in RAID, but any breakpoints set in executing task are ignored. The purpose of this command is to allow the user to monitor global variables as they change with the executions of a task (assuming this is humanly possible). It is best used with relatively infrequently executing tasks. Should a breakpoint be set in an executing task when in RAID, the breakpoint will be ignored and a message outputted indicating this. The message will appear only the first time the particular breakpoint is ignored. The message indicates which break was ignored by printing its rank in the breakpoint table used by RAID. The user may determine which break was ignored by doing a LIST command.

Examples:

```
UN
UNLOCK
```

N. DISPLAY

The DISPLAY command is used to display the value of program variables. The command format is:

```
DISPLAY <variable> [,...,<variable>]
```

This command allows the display of variables local to the procedure from which the entry to RAID was made, GLOBAL variables, and variables referenced under the normal rules of scope. If the procedure from which RAID was entered is one which recursively calls itself, the only local variables which may be seen are those belonging to the most recently invoked version of the procedure. Integer, Real or Character variables may be displayed in scalar or array form. Delay variables may also be displayed.

The type of each variable is determined from symbol information generated by the CRASH compiler. The effect of the type information is shown below:

CHARACTER:
The string presently assigned to the variable is displayed. Each variable is displayed on a separate line. Character strings longer than 132 characters are truncated to the first 132 characters.

REAL: The value of the variable is displayed. Up to 10 values will be printed per line.

INTEGER:
(also BIT variables): The value of the variable is display in base 10. Up to 10 values will be printed per line.

If an array or delay variable name is given, the entire array will be printed in row major order. An option is available to print only a selected range of values in an array or delay variable. Subscript values or ranges may be attached to an array or delay variable reference, e.g. <variable>(lo:hi). In this case, lo and hi specify the range of subscript values for which the array values are to be printed. This range specification works fine for singly dimensioned arrays and delay variables. However, only a restricted form of it is available for arrays of two and three dimensions. Only the lowest order index may have a range as above; higher order indices should appear as single values.

Two options are available for displaying INTEGER or BIT variables in alternate forms. An @B may be attached to a variable specification to cause it to be displayed in binary, and an @O may be attached to cause it to be displayed in octal.

Examples:

```
DISPLAY INT_SCALAR
DIS INT_VEC(5)
D DELAY_VAR(0:3)
DIS TWO_DIM(35,7:10)
DIS THREE_D(4,3,7)
```

```
DIS INT_VAR@N
```

P. MODIFY

The modify command allows values of variables to be changed. The command format is:

```
MODIFY <var_construct> <value>,...,<value>
```

Any variable which may be displayed may be modified. The format for specifying a variable is the same as in DISPLAY. Only a single variable name may be specified per line. The @B and @O modifiers may be used if it is desired to enter new values for INTEGER or BIT variables in binary or octal. New values are separated by one or more spaces. Character strings are enclosed in single primes, e.g. 'a character string'.

Examples:

```
MODIFY INT_VAR 35
MOD REAL_VAR 45.36
M CHAR_VAR 'new string'
MOD INT_VEC(4) 49
MOD REAL_ARRAY(4,2,7:9) 32.1 45.9 98.2
MOD INT_VAR@O 123
```

Q. CALL

The CALL command allows the user to execute an external procedure from RAID. The command format is:

```
CALL <procedure_name>
```

The full procedure name must be entered. At this time, no parameter passing is permitted.

Examples:

```
CALL TASK1
CALL SAMPLE2
```

14.4 Miscellaneous Information

Though RAID is normally entered upon program initiation and upon user set breakpoints, it is possible to use DEBUG to insert IOT instructions in the code, or to insert them via the MTS editor prior to execution of the *11ASR assembler. The IOT instruction will also cause entry to RAID. Caution must be exercised, however. Only CALL, LOCK, UNLOCK, RUN, CONTINUE, or

CLEAN commands can be issued, as RAID has no way of determining from which procedure it was entered, and thus doesn't know which tables to use.

If an error message is printed indicating an error on a command line containing multiple operations, all operations up to the faulty one will have been processed when the error message is printed.

APPENDIX A: RUN-TIME STRATEGY AND CALLING CONVENTIONS.

A.Tasks and Procedures

The task which gets started by the RUN command must be a MAIN procedure. The MAIN procedure will be given a priority of zero (the lowest possible priority) by the RUN command. When the MAIN procedure returns, it and all tasks which it may have started are unloaded and a return is made to the operating system. Similarly, the STOP statement, which can appear in any task, also causes everything to be unloaded and a return to be made to the operating system. A MAIN procedure differs from other external procedures in that it has associated with it an extra csect, #GLOBAL#, which contains all GLOBAL variables. An external procedure is compiled as a separate object module. A TASK is just an external procedure without parameters. If an external procedure is called by an executing task, that invocation of the procedure becomes "part" of the executing task; it may, however, be started as a separate task with an independant priority and execution schedule. All procedures are re-entrant (with the exception of procedures which declare, and change, static variables) since storage for all automatic scalar and array variables will be kept on the stack. Since all procedure parameters are passed by reference rather than by value-result, the procedures are not completely recursive in the usual sense. A user with knowledge of the internal data structures used by CRASH can still obtain recursive behavior however by making judicious use of automatic variables and temporary parameter values which are generated during procedure calls in certain situations.

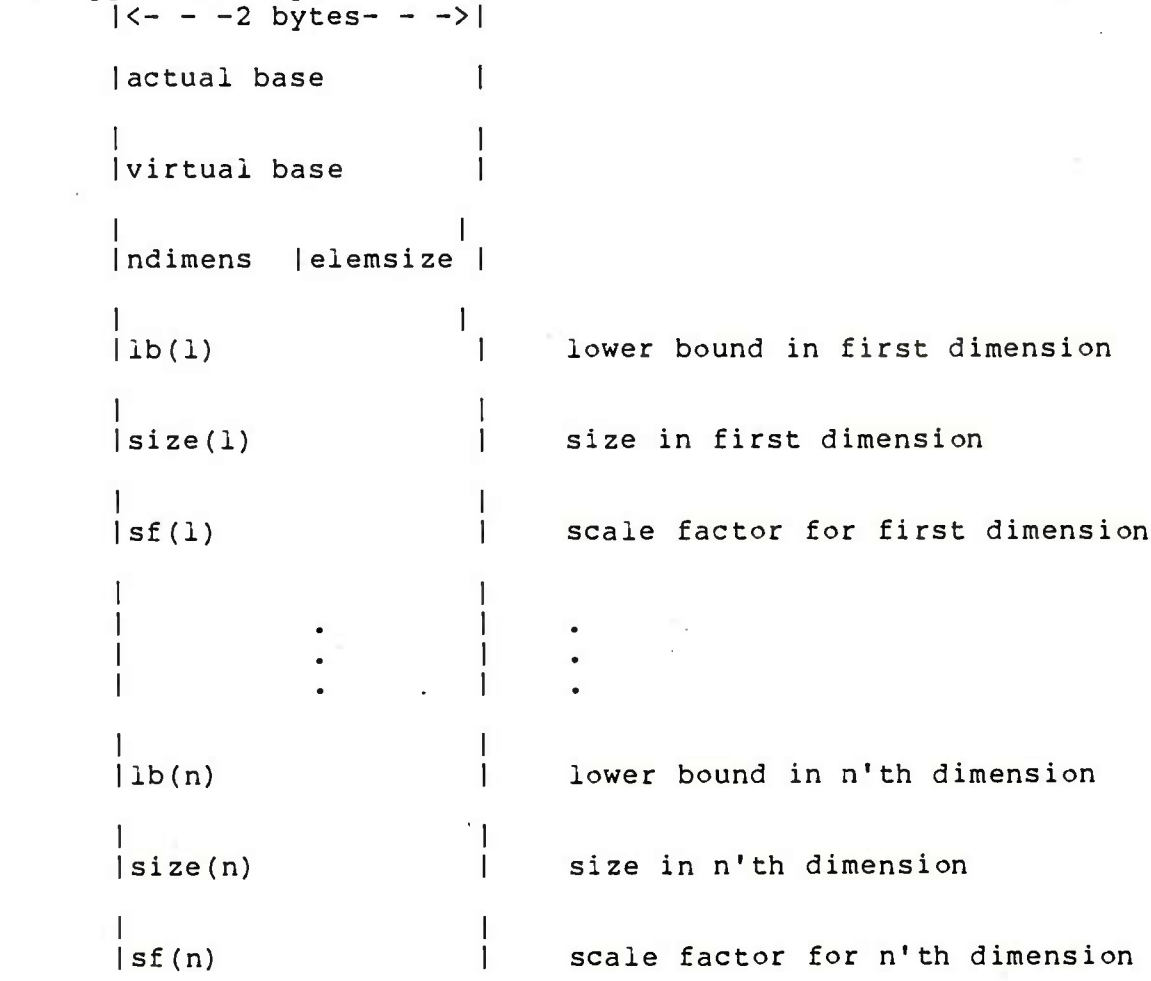
B.Data Types

GLOBAL Variables are implemented as in *EXPL, i.e. as an extra csect. Each global variable then becomes an entry point into the #GLOBAL# csect. A reference to a global variable in a different external procedure is made indirectly through a VCON which will is resolved eventually by the link-editor. As in *EXPL, all global variables must be declared in the MAIN procedure; any initialization for these global variables must also be done in the MAIN procedure. Any global variable declared in another external procedure must have the same type and dimensions; no compile or run time check on this will be performed.

Non-global scalars or arrays can be either STATIC or AUTOMATIC (the default). Storage for a procedure's automatic variables is allocated on the runtime stack by the prologue code upon entry to the procedure---and deallocated by the epilogue code. Storage for all static variables is assigned permanently by CRASH at compile time.

Arrays are stored internally in row major form, as in PL/1,

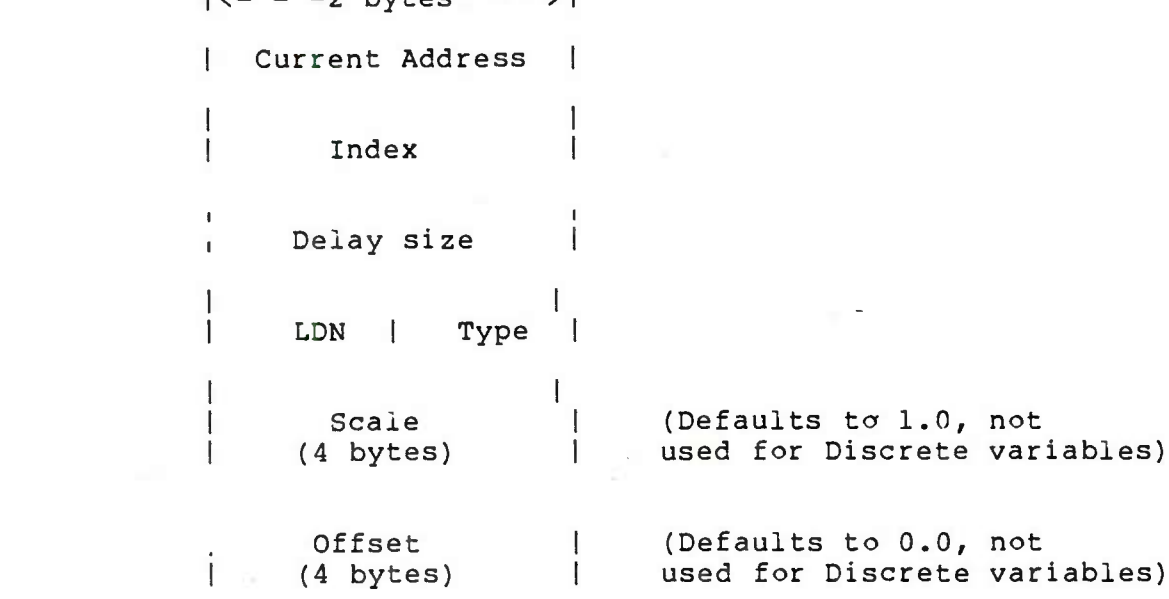
with the rightmost subscript index varying most rapidly. Variable sized arrays are permitted and programmer specified lower bounds are supported. The default lower bound in any dimension will be zero. There will be only one copy of the dope vector for any array, even when it is passed as a parameter to a procedure. As a consequence, arrays must be defined with the same dimensionality in both the calling, and called procedures. To insure compatibility, all formal array parameters to a procedure must have their bounds declared as * (i.e. A(*,*,*)). Arrays can have up to 62 dimensions and may have any upper and lower bounds. The following is a diagram of the dope vector for a typical array:



The address of element I1,I2,...,In is:
Virtual Base + I1 * sf(i)+ I2 * sf(2) + ... + In * sf(n)
Where: Virtual Base = Baseaddr - [lb(1) * sf(1) + ... + lb(n) * sf(n)]
And where: Baseaddr = the address of the array storage area
The array element address calculation will always be done via a submonitor call.

```
sf(i) = size(i+1) * size(i+2) * ... * size(n) * elemsize [i<n]
 sf(n) = elemsize
```

A dope vector is used internally to store all real and integer scalar variables which have the Analog or Discrete attribute. The dope vector looks as follows:



Current Address points to the current delay element.

Index is a counter used in finding the address of a previous (or the next) delay element.

Delay Size is the number of bytes in the delay element storage area: number of delay elements times elementsize (2 or 4).

LDN is the Logical Device Number associated with this variable.

Type tells which type of delay variable it is:

- 0/1 - Discrete
- 2/3 - Real Analog
- 4/5 - Real Analog with default scale & offset
- 6/7 - Integer Analog
- 8/9 - Integer Analog with default scale & offset

Bit 0 specifies Word(0) or Byte(1) I/O.

Scale indicates the scaling associated with this variable.

Offset indicates the offset associated with this variable.

If an Analog or Discrete variable is a procedure parameter, the calling program's dope vector will be used; therefore it is not necessary to declare scales, offsets, etc., for such parameters. If those attributes are provided with a delay variable parameter CRASH will put out a warning message indicating that those attributes will be ignored.

Storage for delay variables is implemented as a circular list. Whenever a new "current" value is input, it is pushed onto the circular list. An uninitialized delay variable dope vector initially contains the last address of the element storage area in "current address" and the elementsize (2 or 4) times the

nnumber of elements to be kept in "index". The procedure for pushing new elements onto the list then is:

1) add the elementsize to both "current address" and "index".

2) if "index" is not greater than "delay size"...finished,

3) else sub "delay size" from both "current address" and "index".

While this algorithm is not the most straight-forward way to implement a circular list, it has the advantage that the corresponding PDP/11 LSI code is very efficient in most cases.

Character Strings are stored with two (fullword aligned) bytes of control information preceding the actual string text. The first byte indicates the maximum length of the string; the second byte indicates the current length of the string. There is no free string area as in *EXPL; a character string occupies its maximum byte size for the duration of its existence. A character string passed as a parameter must be declared with an * size in the called program (i.e. character(*)) to indicate that the size specified by the calling program will be honored.

Note: The format used internally by the operating system for character strings is different. There, character strings are not necessarily full-word aligned, and the string is preceeded only by a one-byte current length --- no maximum length is kept. Whenever CRASH sends the address of a string to the operating system, it adds one to the address it (CRASH) normally uses for the string. Similarly, any string received by CRASH from the operating system must have a maximum length added to it before it can be used by CRASH.

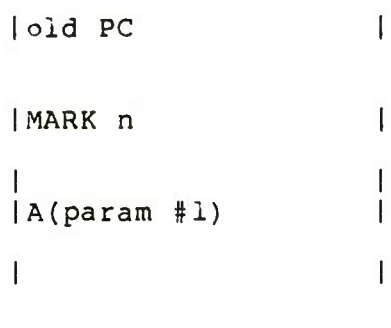
Bit Strings are always stored right-justified in one word (16 bits). A bit string passed as a parameter must also be declared with an * size in the called program (i.e. bit(*)).

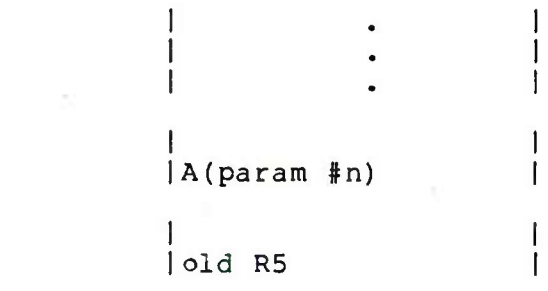
All declarations for BOOLEAN variables in CRASH source programs are transformed in the scanner by predefined macros into declarations for BIT(1) variables.

Integer and Real variables are stored in standard PDP/11 internal form with lengths of 2 and 4 bytes respectively.

C.Calling Sequences

The standard PDP-11 calling conventions will be honored. The stack (upon procedure entry) should look as follows:





R5 points to the MARK instruction. SP points to the old PC. Return to the calling procedure is achieved by issuing an RTS R5 (after resetting SP so that it points to the old PC again). As explained below, R0 and R1 are also used for passing possible function values back from the called program.

The same calling sequence is used for internal and external procedures. The only difference is that an internal procedure is addressed via an ACON while a VCON must be used for external procedures.

Sample Program:

This LSI-11 assembly language program will be called as a CRASH function to add three integers. The operation:

```
RESULT=NUM1+NUM2+NUM3;
```

will be performed by the CRASH call:

```
RESULT=ADDNUMS(NUM1,NUM2,NUM3);
```

where ADDNUMS has been declared as an external integer procedure:

```
INTEGER ADDNUMS EXTERNAL;
```

LSI-11 Program:

```
ADDNUMS     CSECT
            PRINT OFF Supress printing of controls,equ's,etc.
            COPY  K2AT:EQU
            PRINT ON
*
*  EQU's to access our parameters and return value
*
NUM1        EQU   2
NUM2        EQU   4
NUM3        EQU   6
PARBASE     EQU   R5
RETVAL      EQU   R0
*
*  Sum the parameters into R0.
```

```
*
          MOV    NUM1(PARBASE),RETVAL     First parameter
          ADD    NUM2(PARBASE),RETVAL     Plus second parameter
          ADD    NUM3(PARBASE),RETVAL     Plus third parameter
*
*  Now we have NUM1+NUM2+NUM3 in R0, the return value register
*  for INTEGER and BIT functions.Since we haven't used the
*  stack for anything, the stack pointer still points to the
*  OLD PC and we merely:
*
          RTS    R5     To return to caller
          END
```

The above program may be assembled using *11ASR, and then linked with any CRASH program using *LINK11.This would then allow its use as a CRASH function as described.

D.Parameter Passing

All parameters are passed by reference. Procedure name parameters and label parameters are not allowed. There is no implicit conversion of parameters performed during a procedure call.

Function calls and returns are handled as follows:

a) if it is an integer or bit-string function, the value is returned in R0.

b) if it is a real function, the value is returned in R0 and R1.

c) if it's is a string function, at the time of the function call, R0 will contain the address where the result should go.

Any procedure may be called as either a function or as a subroutine. If an integer, real, or bit-string procedure returns a result, the result is put in register(s) R0/R1 and can be ignored. Before calling any procedure, R0 will be zeroed. Then, if a procedure returns a string, it will first examine the address in R0. If that address is non-zero, the string value will be returned; otherwise, no value will be returned.

Variable length parameter lists are also allowed. If more parameters are passed to a procedure than have been declared for it, the extra parameters are ignored. If fewer parameters than have been declared are passed, the remaining parameters are left undefined; any attempt by the procedure to reference these parameters will likely cause trouble. There is a built-in function (NUMARGS) to tell the programmer how many arguments were actually passed.

E.Register Usage

We will follow standard PDP11 conventions on usage of PC, SP and R5 for subroutine calls. During execution of a procedure registers R2-R5 will be used as base registers for the dynamic data areas (DDA's) of other procedures global to the one currently executing. we have permanently reserved R2 for the current level 0 procedure, always an external procedure, and R3-R5 for the current procedures (if any) at levels 1 through 3. As a result of this register allocation scheme, we are not allowing CRASH procedures to be nested more than 4 levels deep --- else we would have to go to a much more complicated register allocation scheme. The remaining registers, R0 and R1, are used as scratch registers; when, rarely, a third scratch register is necessary, one of the base registers is stored temporarily in the DDA at label #ROFLWnn.

The prologue code takes care of setting up the required base registers upon entry to each procedure. USING and DROP pseudo-ops for *11ASR are also issued so that, in the emitted code, variables at any nesting level (with the proper scope rules, of course) can be referenced directly by name.

F.Subscriptrange, Stringrange and Delayoverflow

Subscript checking can be turned on and off for individual arrays at compile time, but this checking is only performed within the procedure currently being compiled. if the array is passed as a parameter to another procedure and subscript checking is desired within that procedure, it must be specified at compile time for that procedure. The same comments also apply to stringrange checking and delay-overflow checking.

G. *11ASR Symbol Name Generation Algorithm

Since CRASH variable names can be up to 255 characters long while the *11ASR assembler allows only a maximum name length of 8 characters, an algorithm must be defined to map CRASH variable names into a form that *11ASR accepts. The *11ASR assembler will be run in batch mode processing the code generated for each external procedure (one CSECT) separately. Therefore, this encoding algorithm need only remove the possibility of conflicts between names generated for any single external procedure and each of its associated internal procedures, if any. There must also be a way for CRASH to define internal names for temporaries, labels, etc. which will not conflict with other CRASH symbol names. The algorithm is as follows:

1) Each procedure is assigned a unique two digit Procedure Number, nn. The external procedure always gets number 00; internal procedures get higher numbers according to their order of definition.
2) The character "#" which is illegal in all CRASH variable names but is legal in *11ASR symbol names, is used in all

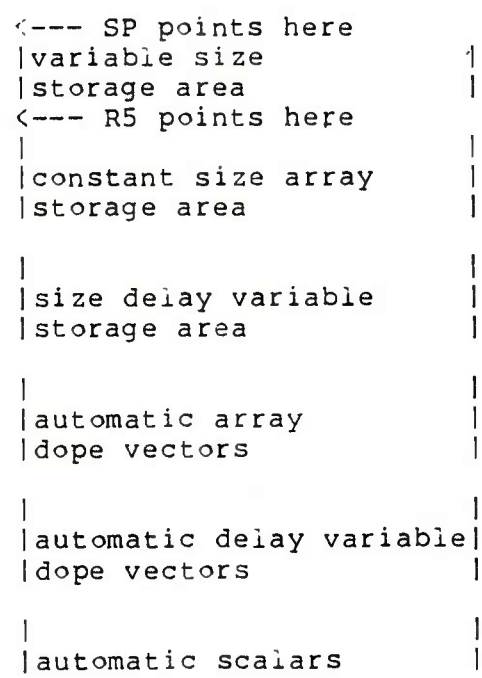
CRASH-generated internal names to prevent conflicts.

3) All CRASH global variables and external procedure names are simply truncated to eight characters (with the illegal *11ASR character "_" replaced by "#") and used directly as *11ASR external symbols. The user must make sure that this truncation does not cause any conflicts.
4) Since the same variable name might be used at several different nesting levels, each static and automatic variable name is modified for *11ASR by appending the first few characters (the number will vary) to #sti (where sti is the index of the variable in the symbol table). Thus if the symbol ERROR_X has index 17 in the symbol table, its generated name will be #17ERROR. If the same symbol is later declared in another procedure (and is located at index 328 in the symbol table) its generated name is #328ERRO.
5) Internal names generated by CRASH consist of "#", followed by a 2-5 letter name (no digits), followed usually by a two digit reference counter. Some of the internal names generated are:

```
#LABii  ---Labels needed for branches
#DDAnn  ---Name used for Dynamic Data Area DSECTS
#STRii  ---String constants
```

H. Storage Allocation

Each internal procedure (and external procedure) will generate one dsect describing a procedure's Dynamic Data Area (DDA) which will look as follows:



```
|                          |
|ON-condition pointers     |
|                          |
|dynamic size              |
|                          |
|register overflow area    |
|                          |
|REGSAVE/REGRSTR area      |
|                          |
|old PC                    |
|                          |
|MARK instruction          |
|                          |
|parameter addresses       |     (closer to address 37777)
```

Description of DSECT Contents:

Variable Size Storage Area: This region is not really part of the DDA. It is allocated after the DDA by the prologue code (discussed later) and is accessed indirectly through array dope vectors which are in the DDA.

Constant Size Array Storage Area: The form is:

```
#AASAii DS ...          automatic array storage area
```

Delay Variable Storage Area: The form is:

```
#DLYSAii DS ...
```

Automatic Array Dope Vectors: The form is:

```
#stiNAME DS A           a full automatic
           .            array dope vector into which
           .            the dope vector skeleton (#SKDVii)
           .            is copied by the prologue code
```

Automatic Delay Variable Dope Vectors: The form is:

```
#stiNAME DS A           a full delay variable
           .            dope vector into which the
           .            dope vector skeleton (#SKDLYii)
           .            is copied by the prologue code
```

Automatic scalars: The form is:

```
#stiNAME DS ...
```

If an initial value was specified, it is moved in by the prologue code.

ON-Condition Pointers: The form is:

```
#DFLTnn DS 2F           pointers for default error checking
#SBCKsti DS F           pointer(s) for subscript checking
                        for each array being checked
           .
           .
#STCKsti DS F           pointer(s) for stringrange checking
                        for each character variable/array
                        being checked
           .
```

```
          .
#DOCKsti DS F          pointer(s) for delay overflow
                       checking for each delay variable
                       being checked
          .
          .
```

This storage is reserved as a result of the presence of CHECK and IGNORE statements. The pointers point to various Submonitor routines which will handle any errors that are found.

Register Overflow Area: The form is:

```
#ROFLW DS F
```

This word is used as a temporary save area for a DDA base register if more than than two scratch registers are required. See the "Register Usage" section for more details.

Dynamic Size: The form is:

```
#DYNSZnn DS F
```

This is the size, in bytes, of the variable size storage area plus the worst case, maximum amount of stack space required. It is set by the prologue code and used for stack overflow checking.

REGSAVE/REGRSTR: This is the save area into which the REGSAVE subroutine stores the values of the registers upon entry to the procedure, and which REGRSTR uses to restore their values at exit. The Regsave subroutine puts the current contents of registers R0-R5 onto the stack, with R5 going on first and R0 last (i.e. R0 is closest to address 00000 and is referenced as #REGSAnn).

The form is:

```
#REGSAnn DS 6F
```

Old PC: Stored automatically during the procedure call by the JSR instruction. The form is:

```
#OLDPCnn DS A
```

Mark Instruction: Stored during the standard subroutine call. It is used to determine the number of parameters actually passed to the subroutine. The form is:

```
#MARKnn DS F
```

Parameters: Parameter addresses are provided by the calling program as specified in the calling conventions. All references to a parameter are made indirectly through the parameter address provided in this area. The form is:

```
#stiNAME DS A
          .
          .
```

#stiNAME is the encoded name of the parameter. The DSECT will contain a label for each parameter, but on a given call, all the parameters may not really be provided by the calling program.

Each external procedure will generate one csect which looks as follows:

```
|procedure body          |    external
|epilogue                |    procedure
|prologue                |    code

|                        |
```

```
|procedure body          |    for first
|epilogue                |    internal procedure
|prologue                |    (if any)

|                        |
|          .             |    other
|          .             |    internal
|          .             |    procedures

|                        |
|string constants        |

|                        |
|static scalar storage   |

|                        |
|static delay variables  |

|                        |
|static array            |
|dope vectors and        |
|array storage           |

|                        |
|automatic array and     |    skeletons for
|delay variable          |    the external procedure
|dope vector skeletons   |    (if any)

|                        |
|          .             |    skeletons for the
|          .             |    internal procedures
|          .             |    (if any)

|                        |
|initial values for      |
|arrays and scalars      |
```

Actually the code for each internal procedure is imbedded in the procedure in which it is defined. The global procedure branches around the internal procedure's code.

The prologue code will be executed first even though it is the last physical part of the procedure code. The inverted order is necessary because CRASH is a one-pass compiler; the information about which ON-Condition pointers are needed, as well as the maximum amount of stack space used (needed for stack overflow checking) cannot be known (& used by the compiler's prologue code emission routines) until the entire procedure has been scanned.

Description of CSECT Contents:

Prologue: (same for external and internal procedures)

a) save the registers on the stack via a call to REGSAVE. The calling sequence for REGSAVE is:

```
MOV   $RGSAVE,-(SP)
JSR   R5,SP)+
```

b) Push SP to leave enough room for fixed portion of the DDA. (check for possible stack overflow first).
SUB size(#DDAnn),SP performs this. Note that size(#DDAnn) can be calculated by the assembler. (i.e. #DDASZnn EQU #REGSAnn-#DDAnn.)
c) Set Rbr to point where SP points (where "br" is the number of the base register for the current DDA).
d) Move in initial values for automatic scalars.
e) Move in automatic array and delay variable dope vector skeletons.
f) Set up automatic delay variable dope vectors.
g) Fill in missing parts of dynamic array dope vectors.
h) Move in initial values for non-dynamic automatic arrays.
i) Acquire room on stack for the variable size array storage (checking for stack overflow first).
j) Initialize the ON-Condition checking pointers to zero
k) Set the Dynamic Size (for stack overflow checking)

The first instruction in the prologue code, the entry point into the procedure, will be labelled with the *11ASR-encoded procedure name which is #stiNAME for internal procedures and the procedure name truncated to 8 characters for external procedures.

Epilogue: (same for external and internal procedures)

a) If this is a character string valued procedure, then if a non-zero return address has been supplied by the calling program, move the return string to that address. Set up the return value in R0/R1 for all other function types. this is done indirectly; the old values of R0/R1 which were saved on the runtime stack by REGSAVE are replaced by these new values for R0/R1. Actually, this step will be done by the code corresponding to the return statement before it emits a branche to the epilogue code.
b) Pop off the variable size array storage area and the DDA from the stack:

```
MOV Rbr,SP          (Rbr is the current DDA base register)
ADD size(#DDAnn),SP
```

c) Restore the registers (via REGRSTR). The calling sequence for REGRSTR is:

```
MOV   $RGRSTR,-(SP)
JSR   R5,SP)+
```

d) Return: RTS R5

The first instruction of the epilogue code will be labelled #EPLGnn. All RETURN statements in the procedure will generate a branch to this label.

String constants: Storage for string constants referenced anywhere within the external procedure or any of the internal procedures. The form is:

```
#STRii DC     H"...",H"...",C'.....'
```

Note that no space is reserved in the CSECT for INTEGER, BIT or REAL constants. Each of these will be assembled as immediate operands. This is also the case with all address constants.

Static Scalar Storage: The form is:

```
    #stiNAME DS ...
```

#stiNAME is an *11ASR name generated using the encoding algorithm described earlier. If an initial value was specified, a DC is used.

Static Delay Variable Dope Vectors and Storage: The form is:

```
    #stiNAME DC F'....'      dope vector
             .               and delay
             .               element
             .               storage
```

Static Array Dope Vectors and Array Storage: The form is:

```
    #stiNAME DC F"virtual base"
             .               same dope
             .               vector format
             .               described earlier
             .
    DS/DC ....               static array storage area
```

#stiNAME is defined as before. The DS/DC saves room for the dope vector and the array and initializes the array storage.

Automatic Array and Delay Variable Dope Vector Skeletons: A separate block of dope vector skeletons is kept here for each procedure; that way, those dope vectors can be moved all at once into the DDA by the prologue code. There are two symbols associated with each block:

```
    #SKCSnn                  label of the block beginning
    #SKCSLnn                 "EQU" for the block length
```

In addition the place into which this data is to be copied in the DDA is labelled "#SKDSnn".

For every automatic array or delay variable, the base address field of the dope vector will always have to be filled in at run time by the procedure entry prologue code. For dynamically dimensioned arrays, more will have to be filled in. The dope vector skeleton is a copy of the dope vector with as much filled in as possible. It is copied to the DDA by prologue code at runtime, and is completed then.

Initial Values for Arrays and Character Scalars: These values are copied into the DDA by the prologue code at procedure entry. The form is:

```
    #INITii DC ...
```

For integer and real scalars, the value need not be stored, but can be initialized by the prologue code using immediate operands. For example: MOV =5,#2VAR ---where VAR has been declared as automatic integer scalar.

I. Submonitor and Operating System Procedures

The following routines are incorporated into the Operating System: Core dump facility LOCK and UNLOCK primitives Data type conversions The following routines will be incorporated into the Submonitor: Array indexing and subscriptrange check Dynamic array dope vector initialization Matrix algebra Mathematical analysis package I routines Substring Concatenation Bit-string manipulations String manipulations Stringrange checking Delay overflow checking Error handling The following functions will be done in-line: Stack

overflow checking NUMARGS (function which returns the number of arguments passed to a procedure) LENGTH (function which returns the current length of a string or character variable) MAXLEN (function which returns the maximum length of a character variable))

APPENDIX B: THE BNF GRAMMAR FOR CRASH

```
$ *
$ * *** GOAL SYMBOL ***
$ *
<PROGRAM> ::= <PROCEDURE LIST>

$ *
$ * *** PROCEDURE DECLARATION ***
$ *
<PROCEDURE LIST> ::= <PROCEDURE DEFINITION> ;
                   |   <PROCEDURE LIST> <PROCEDURE DEFINITION> ;

$ *
$ * *** PROCEDURE DEFINITION ***
$ *
<PROCEDURE DEFINITION> ::= <PROCEDURE DECLARATIONS>
                                        <STATEMENT LIST> <ENDING>

$ *
$ * *** PROCEDURE DECLARATIONS ***
$ *
<PROCEDURE DECLARATIONS> ::= <PROCEDURE HEAD>
                           |   <PROCEDURE HEAD> <DECLARATION LIST>

$ *
$ * *** POSSIBLE HEADS ***
$ *
<PROCEDURE HEAD> ::=  <PROCEDURE NAME> ;
                   |   <PROCEDURE NAME> MAIN ;
                   |   <PROCEDURE NAME> TASK ;
                   |   <PROCEDURE NAME> <PARAMETER LIST>;

$ *
$ * *** PROCEDURE NAME ***
$ *
<PROCEDURE NAME> ::=    <LABEL DEFINITION> PROCEDURE

<PROCEDURE NAME> ::=    <LABEL DEFINITION> <TYPE> PROCEDURE

<LABEL DEFINITION> ::=  <IDENTIFIER> :

$ *
$ * *** PROCEDURE TYPE ***
$ *
<TYPE> ::= INTEGER
         |   REAL
         |   <BIT HEAD> <INTEGER CONSTANT> )
         |   <BIT HEAD> * )
         |   <CHARACTER HEAD> <INTEGER CONSTANT> )
         |   <CHARACTER HEAD> * )
```

```
$ *
$ * *** PARAMETER LIST ***
$ *
<PARAMETER LIST> ::= <PARAMETER HEAD> <IDENTIFIER> )

<PARAMETER HEAD> ::= (
                   |   <PARAMETER HEAD> <IDENTIFIER> ,

<STATEMENT LIST> ::= <STATEMENT>
                   |   <STATEMENT LIST> <STATEMENT>
                   |   <ERROR CHECK> ;
                   |   <STATEMENT LIST> <ERROR CHECK> ;
                   |   <PROCEDURE DEFINITION> ;
                   |   <STATEMENT LIST> <PROCEDURE DEFINITION> ;

$ *
$ * *** TYPES OF STATEMENTS ***
$ *
<STATEMENT> ::=          <BASIC STATEMENT>
              |          <IF STATEMENT>
              |          <SCHEDULE STATEMENT>

$ *
$ * *** BASIC STATEMENTS ***
$ *
<BASIC STATEMENT> ::=    <ASSIGNMENT> ;
                  |      <GROUP> ;
                  |      <RETURN STATEMENT> ;
                  |      <CALL STATEMENT> ;
                  |      <GO TO STATEMENT> ;
                  |      ;
                  |      <GET LIST STATEMENT> ;
                  |      <PUT LIST STATEMENT> ;
                  |      LOCK ;
                  |      UNLOCK ;
                  |      <WAIT STATEMENT> ;
                  |      <EXIT DO> ;
                  |      <NEXT DO> ;
                  |      STOP ;
                  |      <LABEL DEFINITION>  <BASIC STATEMENT>

$ *
$ * *** DECLARATION STATEMENT ***
$ *
<DECLARATION LIST> ::= <DECLARATION STATEMENT>
                   |   <DECLARATION LIST> <DECLARATION STATEMENT>

<DECLARATION STATEMENT> ::= <DECLARE HEAD> <DECLARE ELEMENT> ;

<DECLARE HEAD> ::= <TYPE>
                 |   ROUTINE
                 |   TASK
                 |   <DECLARE HEAD> <DECLARE ELEMENT> ,
```

```
<DECLARE ELEMENT> ::= <IDENTIFIER>
                    |   <IDENTIFIER LIST>
                    |   <DECLARE ELEMENT> <ATTRIBUTE>
                    |   <DECLARE ELEMENT> <DIMENSION LIST>

<IDENTIFIER LIST> ::= <IDENTIFIER HEAD> <IDENTIFIER> )

<IDENTIFIER HEAD> ::= (
                    |   <IDENTIFIER HEAD> <IDENTIFIER> ,

<DIMENSION LIST> ::= <DIMENSION HEAD> <DIMENSION ELEMENT> )

<DIMENSION HEAD> ::= (
                   |   <DIMENSION HEAD> <DIMENSION ELEMENT> ,

<DIMENSION ELEMENT> ::= <DIMENSION BOUND>
                      |   <DIMENSION BOUND> : <DIMENSION BOUND>
                      |   *

<ATTRIBUTE> ::= <LDN HEAD> <INTEGER CONSTANT> )
              |   <DELAY HEAD> <INTEGER CONSTANT> )
              |   <SCALE HEAD> <SIGNED CONSTANT> )
              |   <OFFSET HEAD> <SIGNED CONSTANT> )
              |   <INITIAL HEAD> <SIGNED CONSTANT> )
              |   <INITIAL HEAD> <INTEGER CONSTANT> #
                                                <SIGNED CONSTANT> )
              |   <MAP HEAD> <MAP ELEMENT> )
              |   <CLAMP HEAD> <INTEGER CONSTANT> ,
                                          <INTEGER CONSTANT> )
              |   EXTERNAL
              |   INTERNAL
              |   ANALOG
              |   DISCRETE
              |   GLOBAL
              |   STATIC
              |   WORD
              |   BYTE
              |   PACKED

<BIT HEAD> ::= BIT (

<CHARACTER HEAD> ::= CHARACTER (

<LDN HEAD> ::= LDN (

<DELAY HEAD> ::= DELAY (

<SCALE HEAD> ::= SCALE (

<OFFSET HEAD> ::= OFFSET (

<INITIAL HEAD> ::= INITIAL (
                 |   <INITIAL HEAD> <SIGNED CONSTANT> ,
                 |   <INITIAL HEAD> <INTEGER CONSTANT> #
```

```
                                                    <SIGNED CONSTANT> ,

<MAP HEAD> ::= MAP (
           |   <MAP HEAD> <MAP ELEMENT> ,

<CLAMP HEAD> ::= CLAMP (
             |   <CLAMP HEAD> <SIGNED CONSTANT> ,

<SIGNED CONSTANT> ::= <CONSTANT>
                  |   <+ SIGN> <CONSTANT>
                  |   <- SIGN> <CONSTANT>

<MAP ELEMENT> ::= <IDENTIFIER>
               |   <IDENTIFIER> <FIELD DESCRIPTION>

<FIELD DESCRIPTION> ::= <FIELD HEAD> <EXPRESSION> ]

<FIELD HEAD> ::= [
             |   <FIELD HEAD> <EXPRESSION> ,

$ *
$ * *** ASSIGNMENTS ***
$ *
<ASSIGNMENT> ::=  <VARIABLE> <REPLACE> <EXPRESSION>
              |   <LEFT PART> <ASSIGNMENT>

<REPLACE> ::=     =

<LEFT PART> ::=   <VARIABLE> ,

$ *
$ * *** VARIABLE ***
$ *
<VARIABLE> ::=  <IDENTIFIER>
           |    <SUBSCRIPT HEAD> <EXPRESSION> )
           |    <VARIABLE> @ <SIMPLE VARIABLE>
           |    <VARIABLE> <FIELD DESCRIPTION>

<SUBSCRIPT HEAD> ::=     <IDENTIFIER> (
                 |       <SUBSCRIPT HEAD> <EXPRESSION> ,

$ *
$ * *** DIRECT I/O ***
$ *
<GET LIST STATEMENT> ::= <GET HEAD> <VARIABLE>

<PUT LIST STATEMENT> ::= <PUT HEAD> <VARIABLE>

<GET HEAD> ::= GET
           |   GET RECORD ( <EXPRESSION> )
           |   <GET HEAD> <VARIABLE> ,

<PUT HEAD> ::= PUT
           |   PUT RECORD ( <EXPRESSION> )
```

```
            |   <PUT HEAD> <VARIABLE> ,

$ *
$ * *** WAIT STATEMENT
$ *
<WAIT STATEMENT>  ::= <WAIT> FOR <EXPRESSION>
                  |   <WAIT> FOR <EXPRESSION LIST> <EXPRESSION>

<WAIT>  ::=  WAIT

$ *
$ * *** GO TO STATEMENT ***
$ *
<GO TO STATEMENT> ::=    <GO TO> <IDENTIFIER>

<GO TO> ::=              GO TO
        |                GOTO

$ *
$ * *** CALL STATEMENT ***
$ *
<CALL STATEMENT> ::= <CALL HEAD>
                 |   <CALL HEAD> <CALL PARAMETERS> <EXPRESSION> )

<CALL HEAD> ::=      CALL <IDENTIFIER>

<CALL PARAMETERS> ::= (
                  |   <CALL PARAMETERS> <EXPRESSION> ,

$ *
$ * *** EXPRESSIONS ***
$ *
<EXPRESSION> ::=  <LOGICAL TERM>
             |    <EXPRESSION> OR  <LOGICAL TERM>

$ *
$ * *** LOGICAL EXPRESSIONS ***
$ *
<LOGICAL TERM> ::= <LOGICAL FACTOR>
               |   <LOGICAL TERM> XOR <LOGICAL FACTOR>

<LOGICAL FACTOR> ::=     <LOGICAL SECONDARY>
                 |       <LOGICAL FACTOR> AND <LOGICAL SECONDARY>

<LOGICAL SECONDARY> ::=  <LOGICAL PRIMARY>
                    |    NOT   <LOGICAL PRIMARY>

<LOGICAL PRIMARY> ::=     <TWO BIT EXPRESSION>
                  |       <TWO BIT EXPRESSION> <RELATION>
                                    <TWO BIT EXPRESSION>

<RELATION> ::=           =
           |             <
           |             >
```

```
            |               NOT =
            |               NOT  <
            |               NOT >
            |               < =
            |               > =

$ *
$ * *** TWO BIT EXPRESSIONS ***
$ *
<TWO BIT EXPRESSION> ::= <ARITHMETIC EXPRESSION>
                       |   <ARITHMETIC EXPRESSION> ||
                                        <TWO BIT EXPRESSION>

$ *
$ * *** ARITHMETIC EXPRESSION ***
$ *
<ARITHMETIC EXPRESSION> ::= <TERM>
                          |   <ARITHMETIC EXPRESSION> + <TERM>
                          |   <ARITHMETIC EXPRESSION> - <TERM>
                          |   + <TERM>
                          |   - <TERM>

<TERM> ::=                  <FACTOR>
       |                    <TERM> * <FACTOR>
       |                    <TERM> / <FACTOR>
       |                    <TERM> MOD <FACTOR>

<FACTOR> ::=                <PRIMARY>
         |                  <PRIMARY> ** <FACTOR>

<PRIMARY> ::=               <CONSTANT>
          |                 <VARIABLE>
          |           (  <EXPRESSION> )

$ *
$ * *** CONSTANTS ***
$ *
<CONSTANT> ::= <REAL CONSTANT>
           |   <INTEGER CONSTANT>
           |   <STRING CONSTANT>

$ *
$ * *** GROUP STATEMENTS ***
$ *
<GROUP> ::=                 <GROUP HEAD> <ENDING>

$ *
$ * *** POSSIBLE HEADS ***
$ *
<GROUP HEAD> ::=            <DO> ;
             |              <DO> <STEP DEFINITION> ;
             |              <DO> <WHILE> <EXPRESSION> ;
             |              <DO> <UNTIL> <EXPRESSION> ;
             |              <DO> <CASE> <EXPRESSION> ;
```

```
                |           <GROUP HEAD> <STATEMENT>

$ *
$ * *** STEP DEFINITION ***
$ *
<STEP DEFINITION> ::=  <VARIABLE> <REPLACE> <EXPRESSION>
                                          <ITERATION CONTROL>
                     |   <VARIABLE> <REPLACE> <EXPRESSION LIST>
                                               <EXPRESSION>
                     |   <VARIABLE> <REPLACE> <EXPRESSION>

$ *
$ * *** CONTROL HIS ITERATIONS ***
$ *
<ITERATION CONTROL> ::=  TO <EXPRESSION>
                       |    TO  <EXPRESSION> BY <EXPRESSION>

$ *
$ * *** DO PRODUCTIONS TO HELP SYNTHESIZE ***
$ *
<DO> ::= DO

<UNTIL> ::= UNTIL

<WHILE> ::= WHILE

<CASE> ::= CASE

$ *
$ * *** EXIT DO STATEMENT ***
$ *
<EXIT DO> ::=              EXIT DO
            |              EXIT DO  <IDENTIFIER>

<NEXT DO> ::= NEXT DO
            |   NEXT DO  <IDENTIFIER>

$ *
$ * *** EXPRESSION LIST ***
$ *
<EXPRESSION LIST> ::= <EXPRESSION> ,
                    |   <EXPRESSION LIST> <EXPRESSION> ,

$ *
$ * *** GROUP STATEMENT END ***
$ *
<ENDING> ::=              END
           |              END  <IDENTIFIER>
           |              <LABEL DEFINITION> <ENDING>

$ *
$ * *** RETURN STATEMENT ***
$ *
<RETURN STATEMENT> ::=   RETURN
```

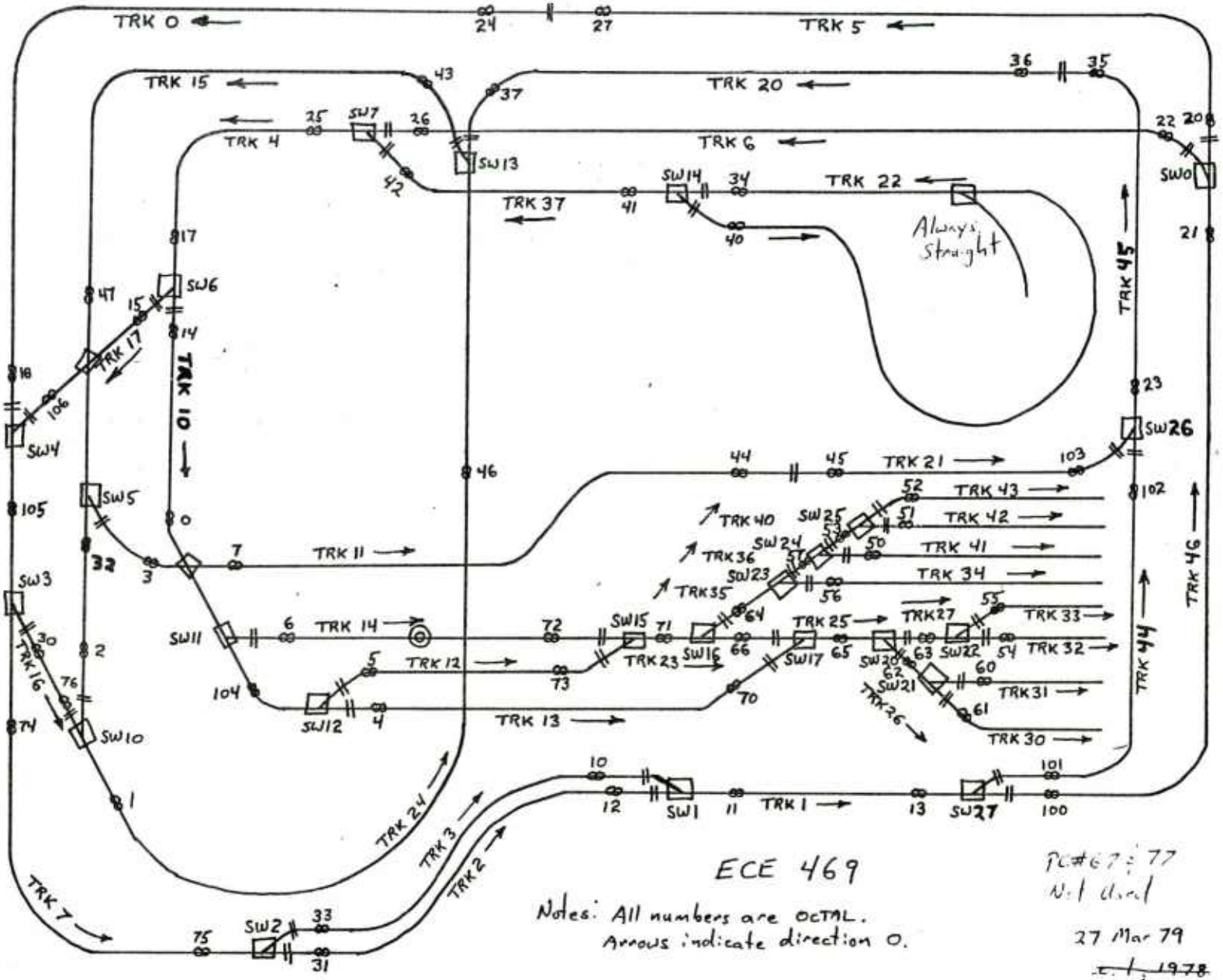
```
                    |     RETURN  <EXPRESSION>

$ *
$ * *** SCHEDULE STATEMENT ***
$ *
<SCHEDULE STATEMENT> ::= <EVERY> <TIME> <START TASK> ;
                       |   <IN> <TIME> <START TASK> ;
                       |   <CANCEL> <TASK> ;
                       |   <ON> <CONDITION> <START TASK> ;
                       |   <ON> <CONDITION> <BASIC STATEMENT>
                       |   <ON> <CONDITION> <IF STATEMENT>
                       |   <AT> <TIME> <START TASK> ;
                       |   <REVERT CONDITION> ;
                       |   <LABEL DEFINITION> <SCHEDULE STATEMENT>
                       |   <START TASK> ;

<CONDITION> ::= <VARIABLE>

<AT> ::= AT

<ON> ::= ON

<IN> ::= IN

<REVERT> ::= REVERT

<CANCEL> ::= CANCEL

<EVERY> ::= EVERY

<TIME> ::= <SIMPLE VARIABLE>
       |   <SIMPLE VARIABLE> MIN
       |   <SIMPLE VARIABLE> SEC
       |   <SIMPLE VARIABLE> MSEC

<TASK> ::= <VARIABLE>
       |   <VARIABLE> <PRIO HEAD> <SIMPLE VARIABLE> )

<PRIO HEAD> ::= PRIO (

$ *
$ * *** COMPILE TIME ERROR CHECK FLAGS ***
$ *
<ERROR CHECK> ::= <CHECK> <CONDITION>
                |   <IGNORE> <CONDITION>

<CHECK> ::= CHECK

<IGNORE> ::= IGNORE

$ *
$ * *** IF STATEMENT ***
$ *
<IF STATEMENT> ::=       <IF CLAUSE> <STATEMENT>
```

```
                |          <IF CLAUSE> <TRUE PART> <STATEMENT>
                |          <LABEL DEFINITION>  <IF STATEMENT>

<IF CLAUSE> ::=    IF <EXPRESSION> THEN

<TRUE PART> ::=    <BASIC STATEMENT> ELSE

$ *
$ * *** JUNK NEEDED TO MAKE (2,1;1,1) PARSING WORK ***
$ *
<+ SIGN> ::= +

<- SIGN> ::= -

<SIMPLE VARIABLE> ::= <IDENTIFIER>
                    |   <INTEGER CONSTANT>
                    |   <REAL CONSTANT>

<DIMENSION BOUND> ::= <SIGNED CONSTANT>
                    |   <IDENTIFIER>
                    |   <+ SIGN> <IDENTIFIER>
                    |   <- SIGN> <IDENTIFIER>

<START TASK> ::= START <TASK>

<REVERT CONDITION> ::= <REVERT> <CONDITION>
```

INDEX *

*Page numbers in this Index refer to the number in the lower corner of each page.

Appendix C: TRAIN LAYOUT



ECE 469

Notes: All numbers are OCTAL.
Arrows indicate direction 0.

PC#67 : 77
N.I dual

27 Mar 79

~~E. I, 1978~~

457

Appendix D: CIRCUITS FOR HARDWARE

Drawings of the hardware circuits are available at the University of Michigan.

Appendix E: DETAILED COURSE OUTLINE

CICE/ECE/IOE 469

Course Outline

Professor Volz

Principles of application of real time computer systems to engineering problems. Topics include: computer characteristics needed for real time use, mini/micro computer operating systems, man-computer communication, basic digital logic design, analog signal processing and conversion, and inter-computer communication. Topics investigated via laboratory using microprocessor system. Three lectures and one three hour laboratory per week.

I. Introduction to Real Time Computations

II. LSI-11 - Software Considerations

- A. Memory organization
- B. Instruction & PSW format
- C. Addressing considerations
- D. Use of assembler and operating system

III. Interrupt Processing and Device Programing - Software Point of View

- A. Introduction to LSI-11 architecture
- B. Priority interrupt processing
- C. Parallel and Serial Interfaces

IV. Conversion Techniques

- A. Simple D/A conversion techniques
- B. Simple A/D conversion techniques
- C. Multiplexing
- D. Typical performance and cost characteristics

V. Data Sampling

- A. Frequency domain viewpoint
- B. Sampling theorem
- C. Practical considerations (e.g. noise and prefiltering)

VI. Introduction to Digital Process Control

- A. Z-transforms and transfer functions
- B. Performance measures
- C. Stability
- D. PID controller
- E. Implementation considerations

VII. Multiple Task Considerations

A. Resource sharing
B. Synchronous and asynchronous tasks
C. Intertask communication
D. Typical "Real Time Operating System" features

VIII. LSI-11 architecture

A. Bus structure
B. Interrupts
C. Direct Memory Access

IX Device Controllers

A. Address decoding
B. Examples of special purpose controllers
C. Noise in logic circuits

X. Computer-Computer Communication

A. Examples of several networks
B. Basic problems in computer networks
C. MCP protocol on MTS

XI. Brief Survey of Other Mini-Micro Computers

A. Architectural differences and impact on performance

Devices controlled by computers in lab

- Servo motor
- Simulations on analog computer
- N guage model railroad

CICE 469

LABORATORY PROBLEM 1

Objective: To familiarize the student with the LSI-11, the CRASH compiler and the OSWIT operating system, and to develop a command parser which can be modified for use in subsequent problems.

Description: A program is to be written which will accept commands of the form:

COM A1 A2 A3 .... AN
where COM is the command and
A1,...AN are 0 or more arguments.
The command and each of the parameters are separated by one or more blanks. A maximum of 10 arguments is to be allowed.

The program must:
1) read a line
2) extract the command
3) accept a proper number (10 max) of arguments
4) check for improper arguments
5) perform the action required by the command.

For this problem, the following commands are to be used:

START
STOP
ENGINE
TRACK
SWITCH
REVERSE
PANIC

The actions to be performed on each command are as follows:

start: No other commands are to be recognized until after a start command has been issued. The command is echoed. No arguments.

panic: Ignore all other commands until the next start command. No arguments.

stop: Terminate program execution. No arguments.

engine: Print the sum of the arguments.

track: Print the mean value of the arguments.

switch: Convert the arguments to floating point and print the product.

reverse: Treat the argument as a character string and print it out in reverse order.

Abbreviations consisting of a sufficient number of the beginning characters of a command to distinguish it from other commands are to be allowed. All commands are to be echoed with the full command name. If arguments appear on the START, STOP or PANIC commands, they are to be ignored.

SAMPLE RUN

```
.RUN    *MTS*                    /* load program from MTS and start*/
EXECUTION BEGINS
?                                /* program is waiting for a command*/
?TRACK  1 2 3                    /* command ignored*/
?START
START                            /* command echoed*/
?E  4 5 6                        /* abbreviated engine command*/
ENGINE 4 5 6
SUM = 15
?S                               /* illegal abbreviation*/
ILLEGAL COMMAND                  /* program message*/
?TRACK  7 8 9                    /* track command*/
TRACK    7 8 9                   /* echo*/
MEAN = 8
?PAN
PANIC
?STOP                            /* command ignored*/
?STA
START
?RE ROSEBOWL
REVERSE ROSEBOWL
```

```
LWOBESOR
?SW   1 5 9                    /* switch command*/
SWITCH   1 5 9
PRODUCT = 45.0
?STOP                          /* end of run*/
STOP
EXECUTION TERMINATED
```

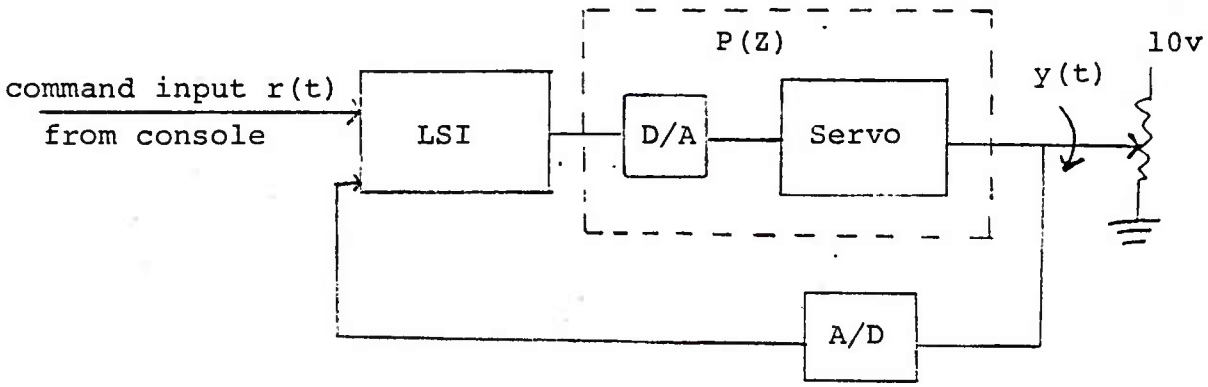
CICE 469

Experiment 3

Objective: To study the effect of two simple controllers in controlling a two phase servo motor system.

1. PID controller
2. position plus velocity feedback controller

Description: A block diagram of the system is shown in Fig. 1.



The output shaft of the servo is coupled to the wiper arm on a potentiometer, which provides an output voltage proportional to the shaft position. This voltage is fed to an A/D converter which provides the input (8 bits) to the computer. Care must be taken to handle wrap around problems. That is 9.9 volts represents an angle only slightly different from that represented by 0.1 volts. Timing is controlled by a programmable real time clock. A variable interval is placed in a clock register. Thus, the basic period of operation can be varied. Each interrupt should trigger the beginning of a new sample period.

The approximate Z-transfer function for the D/A and servo system is

$$P(Z) = K \frac{aT(Z-e^{-aT}) - (1-e^{-aT})(Z-1)}{a^2(Z-1)(Z-e^{-aT})}$$

where

K = gain

1/a = time constant

T = sample period

For this problem K and a are unknown

Implementation: There are several general considerations to be incorporated in your control program:

1. Allow the input value to be set asynchronously from the console.
2. Allow the control parameters to be entered before beginning system operation.
3. Allow selection of control algorithm.
4. Keep sampled values in a circular buffer.
5. Consider D/A problems.
   a) 8 bit overflow
   b) offset
6. Handle wrap around problems.
7. Consider unit conversion.
8. Keep input, output and error history for subsequent printout. e.g. Keep 100 samples after each change in input value.
9. Use zero initial past histories.
10. Design your algorithm to insert a delay of iT seconds in the control where i is an integer in the range [0, 5].
11. Use scaling on K_c, T T_d, T_i, T_v, and u.

PID Controller: The general form of the equation is

$$u[kT] = K_c\{e[kT] + \frac{T}{T_i}\left(\frac{e[0] + e[T]}{2} + \ldots + \frac{e[(k-1)T] + e[kT]}{2}\right) + T_d \frac{e[kT] - e[(k-1)T]}{T}\}.$$

This may be written in a form more convenient for implementation as:

$$u[kT] = u[(k-1)T] + \{K_c + \frac{K_c T}{2T_i} + \frac{K_c T_d}{T}\}e[kT] - \{K_c + \frac{2K_c T_d}{T} - \frac{K_c T}{2T_i}\}e[(k-1)T] + \frac{K_c T_d}{T} e[(k-2)T]$$

where

$$e[kT] = r[kT] - y[kT] \qquad k = 0,1,2,\ldots$$

Velocity feedback controller: The form of this controller is:

$$u[kT] = K_c\{e[KT] - T_v(\frac{y[kT] - y[(k-1)T]}{T})\}$$

Your system is to be command driven and accept at a minimum the following commands (though the commands are shown with an equal sign, this may be omitted if desired):

T=<value>	sets the sample period expressed in ms.
KC=<value>	sets the gain K_C. If integer arithmetic is used it enters $10K_C$.
TD=<value>	sets the derivative coefficient, expressed in ms.
TI=<value>	sets the integral coefficient, expressed in ms.
TV=<value>	sets the velocity feedback coefficient, expressed in ms.
PID	put the controller in PID mode
VF	put the controller in velocity feedback mode.
PR N	print the first N entries of the history table. If N is omitted, print the entire table.
DELAY=<value>	integer value sets the number of periods of time delay.

Programmable clock:

The programmable clock is basically a 32 bit down counter. When a 0 to -1 transition is made the done bit (07) in the CSR is set. If the interrupt enable bit (06) is set, an interrupt will be generated. The done bit is cleared automatically whenever something is written into the upper word of the interval clock.

The addresses are:

CSR:	177020
lower bits:	177022
high order bits:	177024

Experimental runs:

1. Proportional control. Set $T_i = \infty$ and $T_d = 0$ in the PID controller. Vary K_c and T and observe and record the system performance (rise time, overshoot, steady state error). Try to find values of K_c and T for reasonable performance.
2. Proportional plus derivative control. Set $T_i = \infty$ in the PID controller and use the "best" value of T found in 1. Repeat 1 varying K_c and T_d.
3. Velocity feedback controller: Use the sample period of 2 and repeat 1 for varying K_c and T_v.
4. PID control. Repeat 1 for the full PID controller
5. Effect of delay. For any one of the previous controllers insert delays of T, 2T, 3T, 4T and 5T and observe the effect on the response.

Report:

1. Summarize the results of your experimental runs.
2. Endeavor to explain your results in terms of the control theory discussed in class. Try to explain discrepancies with the theory.
3. Include a listing of your program and appropriate code documentation.

CICE 469 PROJECT 2
DATA ACQUISITION

PART 1: COMMAND PARSING

1.1 Purpose

The purpose of part one of this experiment is to further familiarize the student with the LSI-11, the CRASH compiler and the OSWIT operating system by developing a command parser which can be modified for use in part two of this experiment and for subsequent problems.

1.2 Problem Description

A program is to be written which will accept user input commands of the form

```
<COMMAND>[ <ARGUMENT>][<ARGUMENT> ...]
where
   <COMMAND>          is the command
   <ARGUMENT>         is an optional character or numeric
                      value to be supplied with the command
```

The command and each of the arguments are separated by one or more blanks. A maximum of 10 arguments is to be allowed.

For both parts of this problem you will be asked to accept the same set of input commands. Note that the action taken in response to a particular command will be completely different for parts one and two of this problem. The following commands are to be implemented:

```
START
STOP
DSN text-string
FILE file-name
INTERVAL value
NPTS value
DUMP
```

The actions to be performed and required arguments for each command are as follows:

START — No other commands are to be recognized until after a START command has been issued. The command is echoed. No arguments.

STOP — Terminate program execution. The command is echoed. No arguments.

DSN — The argument consists of a character string of arbitrary length. Embedded blanks are to be allowed. The command is echoed, followed by the character string on the next output line.

FILE — The argument consists of a string of nonblank

characters. An argument exceeding ten characters is an error. Any number of blanks may follow the string, but any nonblank character encountered thereafter is an error. The command is echoed followed by the character string enclosed in quotes on one line.

INTERVAL The argument consists of a single integer value in character form. The command is echoed. Convert the argument to integer and print the result.

NPTS The argument consists of a single integer value in character form. The command is echoed. Convert the argument to floating point and print the result.

DUMP Ignore all other commands until the next START command. The command is echoed. No arguments.

Abbreviations consisting of a sufficient number of the beginning characters of a command to distinguish it from other commands are to be allowed. All commands are to be echoed with the full command name for part one, but this is neither required nor desired for part two. If arguments appear on commands which do not require them, you may either ignore them or print a warning. Null lines or lines consisting solely of blanks should be ignored without comment.

1.3 Sample Execution

The following sample terminal session should be used as a guide when designing your program.

```
.RUN *MTS*                          Begin program execution.
.EXECUTION BEGINS.
?DUMP                               (START not yet given)
?START                              User input.
START                               Program output.
?                                   (Ignore null lines)
?DSN sample text string             User input.
DSN                                 Program output.
sample text string                  Program output.
?FILE DATA1                         User input.
FILE "DATA1"                        Program output.
?INT 56                             Abbreviated user input.
INTERVAL                            Output; note full spelling.
    56                              Program output.
?NPTS     32767                     User input.
NPTS                                Program output.
 32767                              Program output.
?DUMP                               User input.
DUMP                                Program output.
?STOP                               (START must now be given)
?STA                                User input.
START                               Program output.
?STOP                               User wants to quit.
.EXECUTION TERMINATED.              So we let him.
```

PART 2: INTRODUCTION TO DATA ACQUISITION

2.1 Purpose

The purpose of part two of this experiment is to introduce some of the problems which arise in the use of computers for on line data acquisition, and to introduce the use of frequency analysis of signals. Some of the questions which arise are: (1)how does one measure the variables in question; (2) how are the values obtained transformed into numbers in the computer; (3) how is the timing of the samples managed; (4) how do you manage volumes of data greater than the main memory size of the computer; (5) and how fast should one sample?

2.2 Problem Description

The issues raised above will be studied in the context of a measurement of respiration rate. The measurement is based on the temperature of the air immediately in front of a subject's nose or mouth. As one breathes out, the temperature is raised, and as one breathes in the temperature drops. A temperature sensitive resistor, called a thermistor, is mounted in a tube through which the subject breathes. When the thermistor is connected in a simple electrical circuit, the voltage across the resistor R will vary as the temperature of the thermistor varies (as the subject breathes). This voltage is then passed through an operational amplifier to adjust the voltage to a range suitable for input to an analog to digital converter. The output of the analog to digital converter may be sampled at any time by the digital computer.

2.2.1 Fast Fourier Transform

Most real time signals are actually composed of an infinite number if infinitesimal sinusoidal signals. The Fourier Transform is the mathematical mechanism which allows the relative amounts of these sinusoids to be determined. In its continuous form, it involves an integration over an infinite time interval, while in its discrete form, it requires a summation over an infinite number of data points. The Fast Fourier Transform is a computationally efficient approximation to the discrete transform which uses only a finite number of points. The efficiency of the computation is dependent upon the number of data points in a very interesting manner. If the number of data points is a power of 2 (512, 1024, 2148, etc) the computation is roughly two and a half times faster than for a nearby number which is not a power of two. Fortunately, library routines exist on MTS to perform the computation.

Let T be the sample period, and let

$$f(0), f(T), f(2T), \ldots, f((n-1)T)$$

be the data points taken. Then the FFT algorithm produces the points

$$F(0), F(w0), F(2w0), \ldots, F((n-1)w0)$$

in the frequency spectrum corresponding to f(t) where

$$w0 = 2 /(n-1)T .$$

These points in the frequency domain are complex. For our purposes, however, only the magnitude is of interest, and we will be working with real numbers. That is, if

$$F(w) = A + jB$$

at some frequency w, we will only be interested in the magnitude

$$\sqrt{A^2 + B^2}.$$

A program will be provided which will take the data samples, perform the FFT calculation, and plot the results. The frequency at which the peak magnitude occurs corresponds to the respiration rate.

2.3 Procedure

For this part of the experiment, you are to modify your command parser as written for the first part to process the required commands in view of the data acquisition procedure. Your problem is to write a suitable program to control the sampling operation and place the sampled data in a file on the floppy disk. The data will then be transmitted, in binary format, to MTS. The discrete Fourier transform of the time samples will then be taken and the respiration rate determined from the peak frequency seen on the frequency spectrum.

2.3.1 Operating Instructions

The controls for the temperature sensor are located in the small box attached to the sensor. The button on that box must be depressed for data to be passed to the A/D converter. If the button is not depressed, then a value of zero is sent to the converter.

Before beginning operation, depress the button and adjust the offset so that breathing through the tube produces a minimal reading of 40 on A/D converter display. Then, when ready to take a set of data samples, type the appropriate command on the terminal, depress the button, and begin breathing through the tube.

2.3.2 Data Sampling Program

Your data sampling program is to be command driven and is to accept at least the following set of commands:

DSN text string

START

STOP

INTERVAL value

NPTS value .

FILE filename

DUMP

The output to your data file is to consist of multiple sets of data. Each data set is to begin with a single line of character data describing the subject, e.g., name, date, time. This line is to be followed by an arbitrary number of data lines containing the samples. The last line in a data set is a trailer record of the form shown below which includes as data the size of the sampling period uses, expressed in milliseconds. The text string on the DSN command is the data to be placed on the character header for the next data set.

The START command is used to start the taking of a set of data. The data sampling is to begin five seconds after the command is entered and is to continue until either the number of points specified in the most recently issued NPTS command is reached, or until a zero sample value occurs, whichever occurs first. This latter condition will occur if the button on the sensor control box is released (which would result in subsequent data values all being zero). The five second delay is to allow time for the user to depress the control button and begin breathing through the tube before the sampling begins.

The STOP command is to terminate execution of the program.

The INTERVAL command is used to set the sample frequency. The integer value associated with the command is to contain the sample period expressed in milliseconds. If this command is not entered, a default value of 100 milliseconds is to be assumed. If the command is entered without a parameter, a command error has occured.

The NPTS command is used to set the number of data points to be taken. The value associated with it contains an integer number of points. If the command is omitted, a default value of 1024 is to be used. If the command is issued without a value, a

command error has occurred.

The FILE command is used to open a file on the floppy for storing respiration data. Design your program so a new data file can be attached without having to rerun your program. If the file cannot be opened an appropriate error message is to be produced.

The DUMP command is used to list the binary contents of the currently opened file on the terminal. Since the respiration data is written as binary bytes, a simple $COPY of the file contents under OSWIT will produce illegible gibberish. Your program is to read each line of the data file, convert to character, and print the data for inspection.

The data sets outputted to the disk file are each to have the following format:

line #	---	contents of line
1	-	header in character form
2	-	data line, format as follows: byte #1 = # of data points on line byte #2 = first data point . . . byte #n = n'th data point
.		
.		more data lines
.		
m	-	m'th data line
m+1	-	3 byte trailer record of the form: 0tt where tt= 2 byte time in msec.
m+2	-	next data set's header if there is another data set following
.		
.		more data sets
.		

Notes:
1) All lines may be longer than necessary with any extra characters ignored.
2) No line can be less than 2 characters in length (MTS/FFT.O restriction)
3) No line can exceed 235 characters (OSWIT/MCP restriction)
4) All data points are in binary format

(and thus the file should be copied
to MTS using the BIN modifier)

The output data file is to contain an arbitrary number of data sets.

2.3.3 FFT Analysis

Copy the data file your have obtained to an MTS file. Then use the FFT.PLOT program to perform the frequency analysis of your data. This is invoked by the following MTS command:

```
$RUN 3ANB:FFT.O SCARDS=DATA.FILE [SPRINT=FREQ.FILE
PAR=FILE=PRINT]
```

DATA.FILE is the file containing your data. The optional FREQ.FILE is an output file which will contain a tabular listing of the magnitudes calculated for the frequencies.

This program should be run online in the computer graphics laboratory with one of the graphics terminals. Since your data files will contain several data sets, the program will pause after plotting eaach of the plots. It will continue after your have hit the return key on the terminal. You should normally make a copy of your plots on the hard copy device before continuing.

Obtain a hardcopy of your data plots and hand them in with your documentation.

2.3.4 Discussion

1. Try to determine the respiration rate directly from a strip chart recording.

2. Compare the results obtained from different breathing experiments.

3. What sources of error do you see in the procedures used? What can you suggest to reduce them?

4. What limitations exist on trying to sample extremely fast for this type of data?

5. The operating system has performed a number of critical tasks for you during this experiment. Identify them and discuss briefly their importance.

2.4 Sanitation

In order to provide sanitary use of the breathing tubes, a disinfectant solution will be kept in the laboratory. After each person has used the tube, it is to be immersed in this solution. The next user is to remove the tube and rinse it thoroughly with cold water before using.

ECE 469

Laboratory Experiment 4

Introduction

The primary objective of this experiment is to provide experience in in dealing with multiple asynchronous interrupts. The setting for this will be computer control of the model train in the laboratory. You may select one of two versions of the problem:

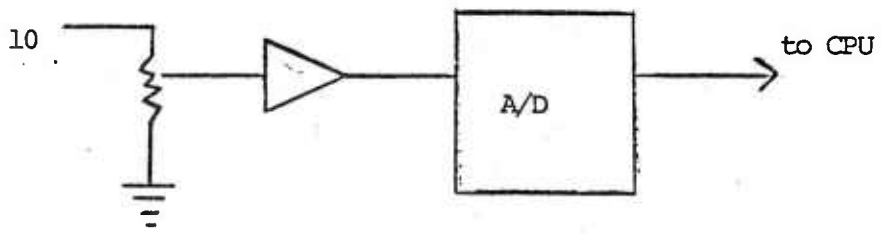
1) control a single engine on the train board.
2) simultaneous independent control of multiple engines.

The latter carries 10% extra credit for the lab. To be eligible to try variation 2) you must receive the permission of your lab instructor, i.e. you must convince him you have a realistic chance of finishing the project. The rest of this write up will be framed as though you were doing variation 2). The items which may be deleted for variation 1) should be obvious.

For each engine to be controlled, there is a throttle and switch. The throttle determines the speed and direction of the engine it controls, and the switch is used to determine the position of track switches the engine approaches. Your system is to handle the setting of track speeds and track switch positions accordingly, and to resolve all conflicts of trains approaching one another or attempting to travel on the same track section at different speeds. In addition to the track speed and track switch controller, two device sensors will be used: photocell sensors to determine train position; and track throttle to determine engine speed. Both of these latter sensors operate through the same parallel interface as the track controller.

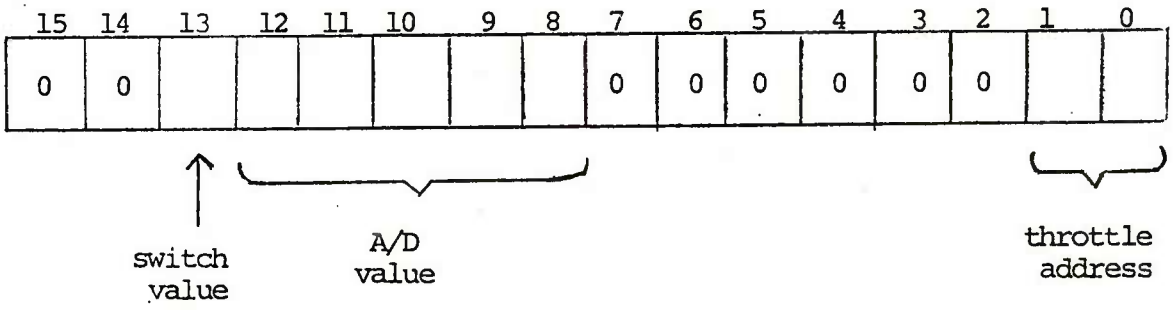
Throttle Control

Each throttle controller is simply a potentiometer connected to a 10 volt source. As shown in Fig. 1, the output is connected to an A/D. You are to treat the center position as zero, with one side being forward, and the other reverse.



Only 5 bits of the 8 in the converter are used. The possible range of values is thus 0 - 31, which must be mapped into an allowable range of -20 to 20. A maximum of four throttles are allowed. Associated with each throttle is a switch used to set whatever track switch the associated engine is approaching.

The A/D converter associated with each throttle is free running. The sensor logic monitors both current and past values from both the switch and the throttle. As long as there is no change in either, no action is taken. Whenever either the throttle or switch shows a change, an interrupt through channel B is generated and the throttle value (0-31) and switch position are placed on the input bus to the parallel interface. The bit usage is shown in Fig. 2.



Photocell Sensor

Photocell sensors are placed around most switches as shown in Fig. 3. The photo cells are adjusted so that normal ambient room light will turn the photocells on. As a train passes over the photocells, they will turn off. The use of the double photocells prevents false indication of end of train caused by light between cars. Each pair has a unique address.

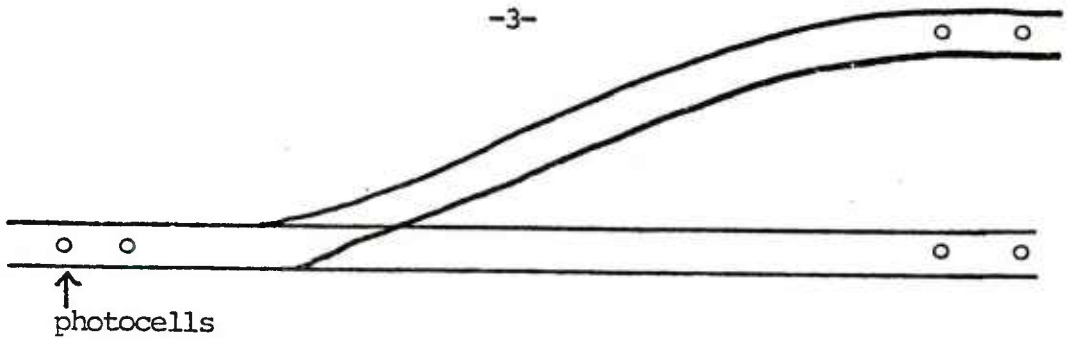


Figure 3.

Whenever one of the photocell pairs changes state (detects a train either entering or leaving the area), an interrupt is generated through the vector address associated with interrupt A on the parallel interface. The photocell address and state are placed in the input buffer- The bit utilization is as shown in Fig. 4.

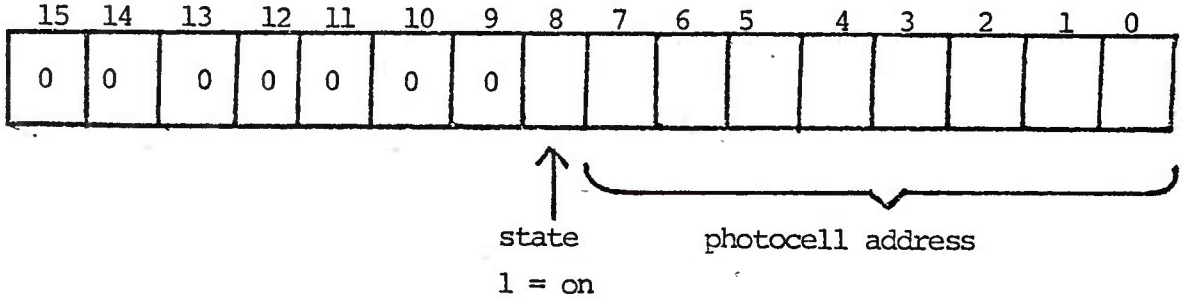
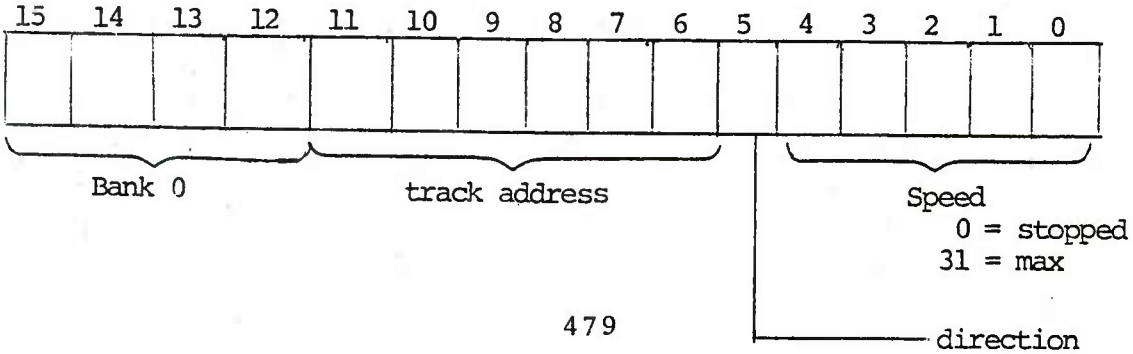


Figure 4.

Track Control

Speed and direction control of an engine is achieved by PUTting an appropriate value to the output unit assigned to the train. The output register is a 16 bit integer whose bits are assigned various functions to effect train control as shown below:



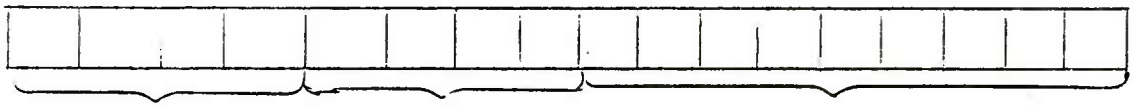
Bits 12-15 select a bank of 64 track sections. At present only bank 0 is implemented. Bits 6-11 address one track in the selected bank. For each track section there is a six bit buffer to hold the assigned direction and speed.

When an integer is PUT to the unit assigned to the train, bits 0-5 are placed in the buffer. Bits 0-4 are then sent to a 5 bit weighted resistor D/A, and bit 5 controls a relay which determines the polarity applied to the track. A 0 in the direction bit will cause an engine to move in the orientation shown by the arrows on the track layout diagram.

Assignment of the selected logical unit to *TRAIN* on the run command will attach the train properly, e.g., 0=*TRAIN*.

Switch Control

Control of the track switches is accomplished through the same unit as the track control by using a different assignment of bits, as shown below:



Bank 15 | Sub-bank | Switch control
0 = straight
1 = turned

Bank 15 is used to indicate switch control rather than track control. Each of the least significant 8 bits controls one switch in the assigned subbank. Note that the switches are then always controlled in groups of 8. The sub-bank assignments are:

sub-bank	switches
0	0-7
1	8-15
2	16-23
3	24-31

Train Control

Your system must input and maintain a data structure describing the track layout. In addition, it will be necessary to maintain certain information on each engine (e.g., current track, next switch position, speed, direction, train position relative to various sensors). To initiate control of a train, the user is to enter through the console the throttle number and a track number upon which an engine has been placed. Initially, the speed for that track is set to zero (regardless of throttle setting).

Any movement of the throttle or switch will then create an interrupt and take control of the train. In addition, there is to be a cancel command which specifies a throttle number and results in removing the train identified from the system.

An important aspect of this problem is the identification of, and definition of decision rules for, various conflicts and situations which can arise. Some of these are:

1. When entering a switch as shown in Fig. 5, the switch should be set to avoid derailment.

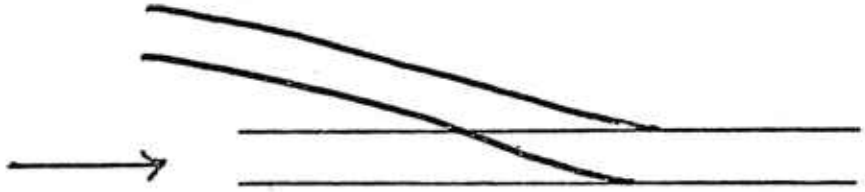


Figure 7.

2. When entering a switch as shown in Fig. 6, the throttle switch must be used to determine the track switch position.

Figure 8.

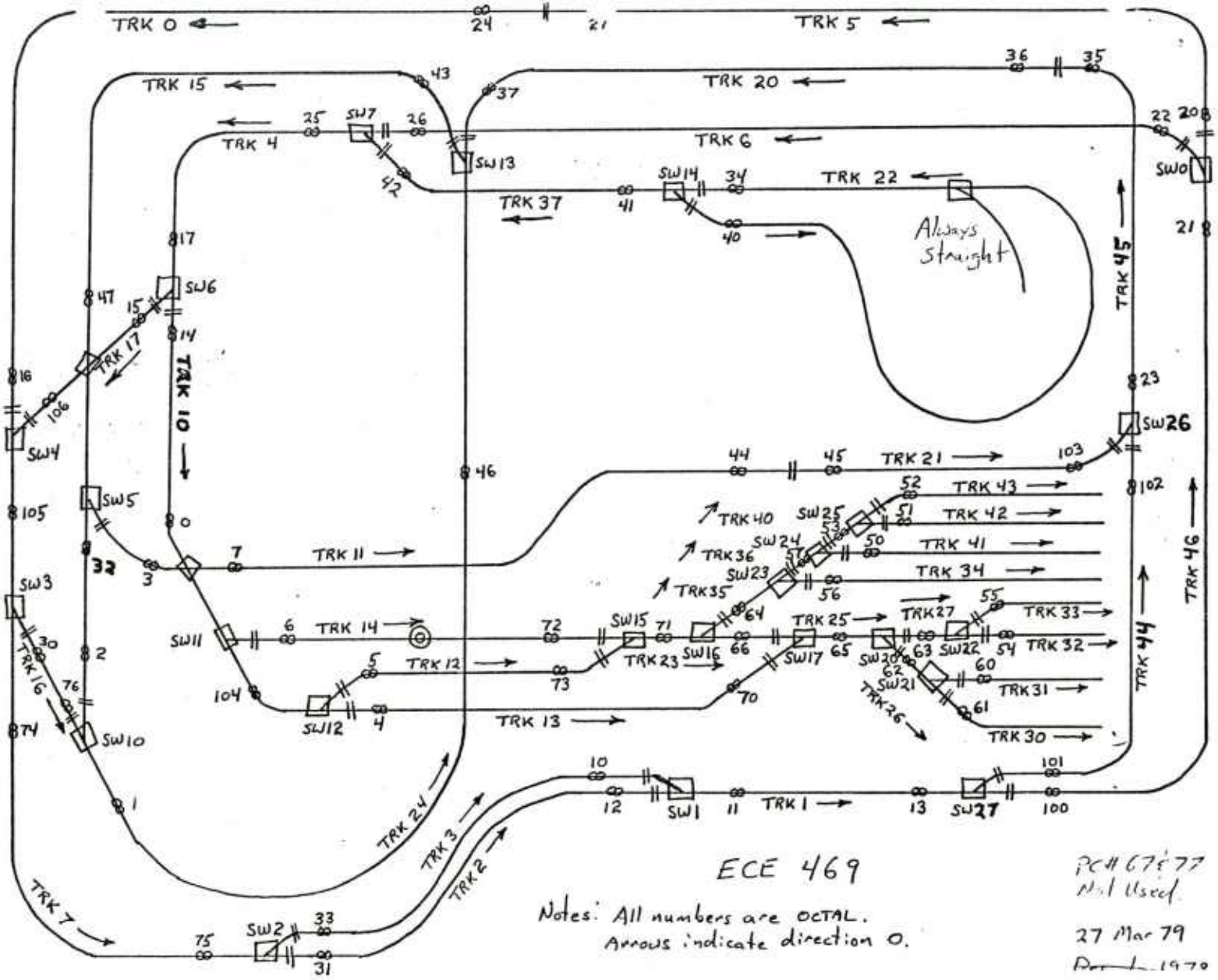
3. When to turn off track power.
4. Resolve collision conflicts.
5. Resolve speed conflicts on same track.
6. Whether or not and how to take into account stationary trains no longer under throttle control (due to cancel).
7. Which track to power up next.
8. When to power up next track.
9. Provide automatic slow down for critical region of track, e.g. where sharp turns or switches are used.

Program Extent

Multiple engine train control is potentially very extensive, possibly more than can be completed in the term. You may state simplifying assumptions to limit the scope of your program, but they should be clearly stated and justification given.

Single engine controls should include at a minimum:

1. Control on all tracks.
2. Control with long trains.
3. Speed limiting to avoid overloading engines.
4. All maneuvers which can be performed on the tracks controlled.
5. Power applied only to track on which the engine resides and, temporarily, to adjacent tracks which may be partially occupied by the train.



ECE 469

Notes: All numbers are OCTAL.
Arrows indicate direction 0.

PCH 6/5/77
Not Used

27 Mar 79
Dec 1979

APPENDIX G: BIBLIOGRAPHY

1. Chuvala, R., P. Beck, "Mechanical Train Analog - A purposed Software Evaluation Tool," ARO Rpt No. 78-3, 1978.

2. Ho, S. B., "A Systematic Approach to the Development and Validation of Software for Critical Applications," Ph.D. Dissertation, University of California, 1977.

3. Stavely, A., "Proving Programs Correct Using Abstract High Level Logic," Ph.D. Dissertation, University of Michigan, 1977.

4. Huang, J. C., "Error Detection Through Program Testing," in Current Trends in Programming Methodology (Vol. II), R. T. Yeh, ed., Englewood Cliffs, NJ, Prentice-Hall, 1977.

5. Goodenough, J. B. and Gerhart, S. L., "Toward a Theory of Test Data Selection," IEEE Transitions on Software Engineering, June 1975.

6. Rose, C. W., "LOGOS and the Software Engineer," AFIPS FJCC, 1972.

7. Burger, R. T., "AUTAS/M: A System for Computerized Assembly of Simulation Models," SIGPLAN Notices, Janury 1974.

INDEX

Metals and Ceramics Information Center
ATTN: Mr. Harold Mindlin, Director
Mr. James Lynch, Asst Director
505 King Avenue
Columbus, OH 43201

Commander
Defense Technical Information
Center (12)
Cameron Station
Alexandria, VA 22314

Commander
U.S. Army Foreign Science and
Technology Center
ATTN: DRXST-SD3
220 Seventh Street NE
Charlottesville, VA 22901

Office of the Deputy Chief of Staff
for Research, Development, and
Acquisition
ATTN: DAMA-ARZ-E
DAMA-CSS
Washington, DC 20310

Commander
Army Research Office
ATTN: Dr. George Mayer
Mr. J. J. Murray
P.O. Box 12211
Research Triangle Park, NC 27709

Commander
U.S. Army Materiel Development and
Readiness Command
ATTN: DRCQA-E
DRCQA-P
DRCDE-D
DRCDMD-FT
DRCLDC
DRCMT
DRCMM-M
Alexandria, VA 22333

Commander
U.S. Army Electronics R&D Command
ATTN: DRSEL-PA-E, Mr. Stan Alster
Mr. Jack Quinn
Fort Monmouth, NJ 07703

Commander
U.S. Army Missile Research and
Development Command
ATTN: DRDMI-TB, Redstone Scientific
Information Center (2)
DRDMI-TK, Mr. J. Alley
DRDMI-M
DRDMI-ET, Mr. R. O. Black
DRDMI-QS, Mr. G. L. Stewart, Jr.
DRDMI-EAT, Mr. R. Talley
DRDMI-QP
Redstone Arsenal, AL 35809

Commander
U.S. Army Troop Support and Aviation
Materiel Readiness Command
ATTN: DRSTS-PLE(2), Mr. J. Corwin
DRSTS-Q
DRSTS-M
4300 Goodfellow Boulevard
St. Louis, MO 63120

Commander
U.S. Army Natick Research and
Development Command
ATTN: DRDNA-EN
Natick, MA 01760

Commander
U.S. Army Mobility Equipment Research
and Development Command
ATTN: DRDME-D
DRDME-E
DRDME-G
DRDME-H
DRDME-M
DRDME-T
DRDME-TQ
DRDME-V
DRDME-ZE
DRDME-N
Fort Belvoir, VA 22060

Commander
U.S. Army Tank-Automotive Materiel
Readiness Command (2)
ATTN: DRSTA-Q
Warren MI 48090

Commander
U.S. Army Armament Materiel
Readiness Command
ATTN: DRSAR-QA (2)
DRSAR-SC
DRSAR-RDP
DRSAR-EN
DRSAR-QAE
Rock Island, IL 61299

Commander
U.S. Army Armament Research and
Development Command
ATTN: DRDAR-LC, Mr. E. Kelly
DRDAR-LCA, Dr. Sharkoff
DRDAR-LCE, Dr. Walker
DRDAR-QAS, Mr. F. Fitzsimmons (5)
DRDAR-SCM, Dr. J. Corrie
DRDAR-TSP, Mr. B. Stephans
DRDAR-LCU-SM, Mr. A. Strano
DRDAR-LCU-SE, Mr. R. Gutter
DRDAR-DP-TD, Mr. Reiter
DRDAR-DP-CP, Mr. Chase
DRDAR-TSS (5)
Dover, NJ 07801

Commander
Edgewood Arsenal
ATTN: DRDAR-CLR, Mr. Montanary
DRDAR-QAC, Dr. Maurits
Aberdeen Proving Ground, MD 21010

Commander
Watervliet Arsenal
ATTN: DRDAR-LCB, Mr. T. Moraczewski
Watervliet, NY 12189

Commander
U.S. Army Aviation R&D Command
ATTN: DRDAV-EXT
DRDAV-QR
DRDAV-QP
DRDAV-QE
St. Louis, MO 63166

Commander
U.S. Army Tank-Automotive Research
and Development Command
ATTN: DRDTA-RKA, Mr. D. Matichuk
DRDTA-RKA, Mr. R. Dunec
DRDTA-RKA, Mr. S. Catalano
DRDTA-JA, Mr. C. Kedzior
DRDTA-UL, Technical Library
Warren, MI 48090

Director
U.S. Army Industrial Base
Engineering Activity
ATTN: DRXIB-MT, Dr. W. T. Yang
Rock Island, IL 61299

Commander
Harry Diamond Laboratories
ATTN: DELHD-EDE, Mr. B. F. Willis
280 Powder Mill Road
Adelphi, MD 20783

Commander
U.S. Army Test and Evaluation Command
ATTN: DRSTE-TD
DRSTE-ME
Aberdeen Proving Ground, MD 21005

Commander
U.S. Army White Sands Missile Range
ATTN: STEWS-AD-L
STEWS-ID
STEWS-TD-PM
White Sands Missile Range, NM 88002

Commander
U.S. Army Yuma Proving Ground
ATTN: Technical Library
Yuma, AR 85364

Commander
U.S. Army Tropic Test Center
ATTN: STETC-TD, Drawer 942
Fort Clayton, Canal Zone

Commander
Aberdeen Proving Ground
ATTN: STEAP-MT
STEAP-TL
STEAP-MT-M, Mr. J. A. Feroli
STEAP-MT-G, Mr. R. L. Huddleston
Aberdeen Proving Ground, MD 21005

Commander
U.S. Army Cold Region Test Center
ATTN: STECR-OP-PM
APO Seattle, Washington 98733

Commander
U.S. Army Dugway Proving Ground
ATTN: STEDP-MT
Dugway, UT 84022

Commander
U.S. Army Electronic Proving Ground
ATTN: STEEP-MT
Ft. Huachuca, AZ 85613

Commander
Jefferson Proving Ground
ATTN: STEJP-TD-I
Madison, IN 47250

Commander
U.S. Army Aircraft Development
Test Activity
ATTN: STEBG-TD
Ft. Rucker, AL 36362

President
U.S. Army Armor and Engineer Board
ATTN: ATZKOAE-TA
Ft. Knox, KY 40121

President
U.S. Army Field Artillery Board
ATTN: ATZR-BDOP
Ft. Sill, OK 73503

Commander
Anniston Army Depot
ATTN: SDSAN-QA
Anniston, AL 36202

Commander
Corpus Christi Army Depot
ATTN:SDSCC-MEE, Mr. Haggerty
Mail Stop 55
Corpus Christi, TX 78419

Commander
Letterkenny Army Depot
ATTN: SDSLE-QA
Chambersburg, PA 17201

Commander
Lexington-Bluegrass Army Depot
ATTN: SDSRR-QA
Lexington, KY 405a07

Commander
New Cumberland Army Depot
ATTN: SDSNC-QA
New Cumberland, PA 17070

Commander
U.S. Army Depot Activity, Pueblo
ATTN: SDSTE-PU-Q (2)
Pueblo, CO 81001

Commander
Red River Army Depot
ATTN: SDSRR-QA
Texarkana, TX 75501

Commander
Sacramento Army Depot
ATTN: SDSSA-QA
Sacramento, CA 95813

Commander
Savanna Army Depot Activity
ATTN: SDSSV-S
Savanna, IL 61074

Commander
Seneca Army Depot
ATTN: SDSSE-R
Romulus, NY 14541

Commander
Sharpe Army Depot
ATTN: SDSSH-QE
Lathrop, CA 95330

Commander
Sierra Army Depot
ATTN: SDSSI-DQA
Herlong, CA 96113

Commander
Tobyhanna Army Depot
ATTN: SDSTO-Q
Tobyhanna, PA 18466

Commander
Tooele Army Depot
ATTN: SDSTE-QA
Tooele, UT 84074

Director
DARCOM Ammunition Center
ATTN: SARAC-DE
Savanna, IL 61074

Naval Research Laboratory
ATTN: Dr. J. M. Krafft, Code 8430
Library, Code 2620
Washington, DC 20375

Air Force Materials Laboratory
ATTN: AFML-DO, Library
AFML-LTM, Mr. E. Wheeler
AFML-LLP, Mr. R. Rowand
Wright-Patterson AFB, OH 45433

Director
Army Materials and Mechanics
Research Center
ATTN: DRXMR-P
DRXMR-PL (2)
DRXMR-M (2)
DRXMR-MQ
DRXMR-MI, Mr. Darcy
DRXMR-L, Dr. Chait
DRXMR-RA, Mr. Valente
DRXMR-AG-MD
DRXMR-X
DRXMR-PR
Watertown, MA 02172

Weapon System Concept Team/CSL
ATTN: DRDAR-ACW
Aberdeen Proving Ground, MD 21010

Technical Library
ATTN: DRDAR-CLJ-L
Aberdeen Proving Ground, MD 21010

Director
U.S. Army Ballistic Research Laboratory
ARRADCOM
ATTN: DRDAR-TSB-S
Aberdeen Proving Ground, MD 21005

Benet Weapons Laboratory
Technical Library
ATTN: DRDAR-LCB-TL
Watervliet, NY 12189

Commander
U.S. Army Armament Materiel
Readiness Command
ATTN: DRSAR-LEP-L
Rock Island, IL 61299

Director
U.S. Army TRADOC Systems Analysis
Activity
ATTN: ATAA-SL (Technical Library)
White Sands Missile Range, NM 88002

U.S. Army Materiel Systems
Analysis Activity
ATTN: DRSXY-MP
Aberdeen Proving Ground, MD 21005